
AFRL-RI-RS-TR-2014-083

# A FORMAL APPROACH TO THE PROVABLY CORRECT SYNTHESIS OF MISSION CRITICAL EMBEDDED SOFTWARE FOR MULTI CORE EMBEDDED PLATFORMS

VIRGINIA POLYTECHNIC INSTITUTE

*APRIL 2014*

FINAL TECHNICAL REPORT

***APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED***

STINFO COPY

**AIR FORCE RESEARCH LABORATORY**
**INFORMATION DIRECTORATE**

■ **AIR FORCE MATERIEL COMMAND** ■ **UNITED STATES AIR FORCE** ■ **ROME, NY 13441**

# NOTICE AND SIGNATURE PAGE

Using Government drawings, specifications, or other data included in this document for any purpose other than Government procurement does not in any way obligate the U.S. Government. The fact that the Government formulated or supplied the drawings, specifications, or other data does not license the holder or any other person or corporation; or convey any rights or permission to manufacture, use, or sell any patented invention that may relate to them.

This report is the result of contracted fundamental research deemed exempt from public affairs security and policy review in accordance with SAF/AQR memorandum dated 10 Dec 08 and AFRL/CA policy clarification memorandum dated 16 Jan 09. This report is available to the general public, including foreign nationals. Copies may be obtained from the Defense Technical Information Center (DTIC) (http://www.dtic.mil).

AFRL-RI-RS-TR-2014-083 HAS BEEN REVIEWED AND IS APPROVED FOR PUBLICATION IN ACCORDANCE WITH ASSIGNED DISTRIBUTION STATEMENT.

FOR THE DIRECTOR:

**/ S /**
WILLIAM MCKEEVER
Work Unit Manager

**/ S /**
MARK H. LINDERMAN
Technical Advisor, Computing
& Communications Division
Information Directorate

This report is published in the interest of scientific and technical information exchange, and its publication does not constitute the Government's approval or disapproval of its ideas or findings.

# REPORT DOCUMENTATION PAGE

*Form Approved*
**OMB No. 0704-0188**



| 1. REPORT DATE *(DD-MM-YYYY)* | 2. REPORT TYPE | 3. DATES COVERED *(From - To)* |
|---|---|---|
| APRIL 2014 | FINAL TECHNICAL REPORT | OCT 2010 – OCT 2013 |

**4. TITLE AND SUBTITLE**

A FORMAL APPROACH TO THE PROVABLY CORRECT SYNTHESIS OF MISSION CRITICAL EMBEDDED SOFTWARE FOR MULTI CORE EMBEDDED PLATFORMS

**5a. CONTRACT NUMBER**

**5b. GRANT NUMBER**
FA8750-11-1-0042

**5c. PROGRAM ELEMENT NUMBER**

**6. AUTHOR(S)**

S. Shukla, M. Nanjundappa, M. Anderson, B. Jose, M. Kracht, and J.Ouy

**5d. PROJECT NUMBER**
T2CS

**5e. TASK NUMBER**
VT

**5f. WORK UNIT NUMBER**
CC

**7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)**
Virginia Polytechnic Institute
1880 Pratt Drive STE 2006
Blacksburg, VA 24060-6750

**8. PERFORMING ORGANIZATION REPORT NUMBER**

**9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)**

Air Force Research Laboratory/RITA
525 Brooks Road
Rome NY 13441-4505

**10. SPONSOR/MONITOR'S ACRONYM(S)**
AFRL/RI

**11. SPONSOR/MONITOR'S REPORT NUMBER**
AFRL-RI-RS-TR-2014-083

**12. DISTRIBUTION AVAILABILITY STATEMENT**
Approved for Public Release; Distribution Unlimited. This report is the result of contracted fundamental research deemed exempt from public affairs security and policy review in accordance with SAF/AQR memorandum dated 10 Dec 08 and AFRL/CA policy clarification memorandum dated 16 Jan 09.

**13. SUPPLEMENTARY NOTES**

**14. ABSTRACT**

This is the final report on the findings of the AFRL funded project "A Formal Approach to the Provably Correct Synthesis of Mission Critical Embedded Software for Multi-core Platforms". In this work we enhanced the theory of a formal modeling language based specifications, namely MRICDF. We demonstrated an implementation of a software specification and code synthesis tool based on MRICDF. The work entails new synthesis algorithms, characterization of specifications, formal proof techniques for proving the correctness preservation property of the refinement steps in our step-wise refinement oriented synthesis technique, multi-core code synthesis, endowing the specification with platform specific worst case execution times to check real-time schedulability, and some case studies.

**15. SUBJECT TERMS**
Software Engineering, Software Producibility, Component-based software design, behavioral types, behavioral type inference, Polychronous model of computation, Prime Implicates, Boolean Abstraction, real-time embedded software, software synthesis, correct by construction software design, model-driven software design, high-assurance software

| 16. SECURITY CLASSIFICATION OF: | | | 17. LIMITATION OF ABSTRACT | 18. NUMBER OF PAGES | 19a. NAME OF RESPONSIBLE PERSON |
|---|---|---|---|---|---|
| a. REPORT | b. ABSTRACT | c. THIS PAGE | | | **WILLIAM McKEEVER** |
| U | U | U | UU | 94 | 19b. TELEPHONE NUMBER (*Include area code*) **315-330-2897** |

# Table of Contents

# List of Figures



# List of Tables

# Foreword

Software assurance for safety-critical systems is an increasingly difficult problem as software systems are becoming more complex, and ubiquitous. From the perspective of critical infrastructures, civilian or military, almost every aspect of all systems are now software controlled, be it the drive-by-wire automotive, military vehicles, fly-by-wire fighter jets or civil aviation, unmanned vehicles and drones, control of a manufacturing plant, or a power plant, or medical devices implanted in patients. Ensuring correctness of software that are going to be used for real-time control of so many safety-critical systems is a complex problem, and one can take a two prong approach to it. One approach is post-facto verification (testing, modeling checking, theorem proving based proof of correctness etc.). The other approach is to enable methods for correct-by-construction software synthesis from mathematically sound specifications. It turns out that for historical and cultural reasons, the first approach has gotten more research and educational attention in the United States in the last 20-30 years, whereas in Europe, correct-by-construction synthesis has had at least two decades of effort, and educational training. Therefore, it is quite reasonable to see that most successful software verification tools (although limited in capacity and scope) have originated in the US, whereas most well-known synthesis tools and methodologies are from Europe. For example, the SPIN model checker developed in the AT&T Bell Labs for verifying software used in telephone call routing has evolved into a software verification tool at the Jet Propulsion laboratory in Pasadena California. Since most successful semiconductor industries are in the US, the hardware formal verification tool vendors are also US centric (Cadence, Jasper Design Automation, and Prover Technologies etc.). On the other hand, SCADE is a synthesis tool that is used by Air Bus, and other companies in Europe and even at General Electric in the US, and it originated in France. Even though SCADE is very limited in scope, and it can only be used for a very specific kind of control loop synthesis, the commercial success of SCADE is very Europe Centric.

Fortunately, since 2001, I have been involved in joint research with the French National Institute for Computer Science and Automation (INRIA) in their software synthesis methodology based on Polychronous specification language SIGNAL, and the corresponding tool suite. This specification formalism is much more powerful when concurrency has to be modeled at the specification level (SCADE is a synchronous language and handles concurrency poorly by over synchronization of concurrent computation). In order to bring that technology to the US Air Force, I started developing an alternative polychronous specification formalism called MRICDF (Multi-Rate Instantaneous Channel Connected Data Flow Network), and the corresponding visual language, and modeling/synthesis tool called EmCodeSyn. This work originated when I spent a summer with Steve Drager and William McKeever at the Air Force Rome Labs during the summer of 2008. Immediately after that I spent a sabbatical year in France and in Germany, in two groups at two institutes attempting two different approaches to software synthesis, I got this idea of developing the theory, and tools for software synthesis for concurrent multi-threaded applications. Since around 2008-09, the multi-core processors started to become common place even on our desktops, it was an imperative to develop tools and frameworks for synthesis of code for such platforms.

However, since the software for embedded safety-critical system are often time constrained, and we need real-time multi-threaded software running on a real-time operating system, the further enhancement of EmCodeSyn that we planned for was to develop real-time EmCodeSyn,

and the corresponding modeling framework, and the schedulability checker, and scheduled code synthesis.

We also planned to enhance EmcodeSyn with some automated redundancy insertion techniques for dependability, but due to the budget cut brought on by the sequestration in 2013, we had to fall short on that.

# Preface

Model-driven embedded software synthesis is an area of research for a while in various forms. For example, Mathworks Inc, has a tool 'real-time workshop' that can generate C/C++ code from Simulink/MatLab specifications. However, since Simulink/Matlab does not have a published formal semantics, we are unable to formally guarantee that the generated code is correct with respect to the specification. The notion of correctness being that the set of infinite behaviors of the generated code is exactly the set of infinite behaviors of the original model. Therefore, in many industries, including Boeing, the real-time workshop tool is used to generate code only to treat the code as hand-written code, and hence all verification efforts remain the same. Moreover, since auto-generated code might not be easy to read, it creates extra burden on the verification engineers. As a result, for a 'correct-by-construction' synthesis, one needs to start with a specification that is formal. The specification language must have precise formal semantics, so that the meaning (in terms of the set of behaviors implied by the model) is unambiguous. Moreover, having started with a formal specification, one can use step-wise refinement techniques to transform the model into an implementation. The process of doing so is called 'software synthesis'. The synthesis process (also called the compilation process) must be proven to preserve all the properties of the model. Thus, if the model can be proven to possess all requisite properties, one can then assume that the implementation preserves all those properties. Proving correctness of the implementation is often more cumbersome, as the implementation often contains details, programming language specific idiosyncrasies, and often too large to prove correct with today's formal verification tools. As a result, this 'correct-by-construction' approach, if it works, can gain much over the traditional post-implementation testing and verification.

Other than informal modeling languages such as Simulink, Labview, Modelica etc, there have been a lot of effort in Europe in defining languages that can model embedded reactive software systems. Synchronous languages such as Esterel, Lustre, Quartz, Argos, etc., are the results of those efforts. Even though, these languages differ syntactically, and in their implicit model of computation, they all simplify embedded computing into a sequence of reactions. The idea is that there are sensors through which the embedded controller samples the physical state of the system it is controlling, using the measurements from the sensors it computes the present state, and use a state machine algorithm to decide what state should the controller be in next, and what actuations must be done on the physical system to bring it to the desired state. This entire process can be thought of as a reaction. In the synchrony hypothesis underlying all these different languages assumes that the time taken for one reaction is small enough so that the when the next inputs are obtained, the reaction has already taken place. Therefore, one can abstract the entire reaction time to a single logical instant. Thus, the entire computation can be thought as a totally ordered sequence of reactions. When to start a new reaction, and when it is time to finish the reaction is determined by some periodic clock (usually the clock period must be at least as large as the worst case reaction time).

This model is simple enough, that the Lustre language has given rise to the popular tool SCADE that even regular engineers can use to specify a control loop, and then generate the code. However, this model of time is called 'globally linearized time' or 'global-clock' based computation. What happens if the reaction requires multiple components who actually communicate over a network, and the system has unspecified delays on the network. If components are placed in such asynchronous environment, there is no way to guarantee that the

components will all compute in unison and finish the reaction before the next global clock ticks. Unfortunately, most modern embedded systems are highly concurrent and distributed, and hence such synchronization of all the components can be expensive, and inefficient for performance. Thus, one requires a modeling formalism where the components do not need to synchronize in all reactions, and some of the time, the components can go on computing asynchronous to the other components, and synchronize when they need to agree on a value, or participate in a computation together. This kind of computation is more common for today's distributed embedded systems, and certainly for multi-threaded embedded software. Thus, the notion of Polychronous model of time was invented in early 1990s in France, and the corresponding language SIGNAL, and a tool set called Polychrony was developed in the last 20 years.

However, since the formalism of polychrony is no longer simple linear sequence of reactions, and the components must decide when to compute without synchronizing with other components, and when to wait for synchronization, the entire mathematical machinery of this formalism involves partial ordered notion of time, partial order of reactions, and partial order of logical instants. This is quite cumbersome for regular engineers to use, and hence unlike Lustre, which has been popularized by the prevalence of the SCADA tool in the industry, Polychrony did not gain the popularity it deserved as it is surely the right model of computation for what we want.

Having worked with the inventors of the polychrony over 10 years, I decided that we can do better semantic framework for polychrony in terms of a Boolean abstraction, and theory of Prime Implicates which was unknown the inventors of Polychrony, and we can also enhance the formalism in many different ways, for better code generation, checking for code synthesizability, and also provide methods for real-time code synthesis. This resulted in our version of Polychrony – which we named MRICDF as we took a more visual approach to the modeling language, and we also developed a visual framework for modeling, and code synthesis, specification of real-time task structures, and code synthesis for real-time applications etc.

In this report, we will provide the details on the MRICDF language, our alternative semantics that enabled not only sequential code synthesis when possible, but also algorithmic tests for sequential implementability, our multi-threaded code synthesis, and real-time code synthesis.

Note that, one important aspect for safety-critical concurrent systems is that we need to synthesize code that is deterministic. If the code could be non-deterministic, then the problem is easier, but no software assurance can be given. Thus much of the complexity of the model-driven code synthesis – be it for synchronous programming, or polychronous programming is germane in the need for determinism.

While working on EmCodeSyn, we also considered the entire system design problem, rather than just the controller software modeling and synthesis. As a result, we started a new approach enabled by AADL and MRICDF called APECS, which allow us to formally capture the entire system platform (hardware and software architecture), and use MRICDF as software component specifications, and then use the EmCodeSyn code synthesis for the system level software synthesis. Even though, we did not plan on this topic in the original project proposal, this provides us a Segway from the component level software design and synthesis problem to the system level design problem, and an initial approach to solving the entire system design problem.

## Acknowledgement

We acknowledge the support of William McKeever, and Steve Drager from the Air Force Rome Laboratory (AFRL) throughout the project. Besides funding this project, they have continuously provided feedback and helped us in many different technical and nontechnical issues related to this project.

The ESPRESSO research team at the French National Institute of Computer Science and Automation (INRIA) – particularly Jean-Pierre Talpin has provided valuable advice and feedback on the use of Polychrony and polychronous model of computation. Jens Brandt from the Quartz project at the Technical University of Kaiserslautern, Germany also helped in many ways - as the theme of this project is also of great interest in Europe.

# 1 Summary

Our past interactions with the U.S. Air Force Labs, Boeing, and Lockheed Martin indicate that embedded software is mostly programmed manually even today. Even when synthesized from MATLAB/Simulink, the code is not provably correct, and expensive verification is required. Anyone experienced with multithreaded programming would recognize the difficulty of designing and implementing such software. Resolving concurrency, synchronization, and coordination issues, and tackling the non-determinism germane in multi-threaded software is extremely difficult. Ensuring correctness with respect to the specification and deterministic behavior is necessary for safe execution of such code. It is therefore desirable to synthesize multi-threaded code from formal specifications using a provably 'correct-by construction' approach. In Europe, it has been widely claimed that the embedded software for 'fly-by wire' control of the AirBus-380 was mostly automatically generated using SCADE and other French tools based on the synchronous programming models. Unfortunately, software generated in those contexts usually operates in a time-triggered execution model. Such models are simpler but less efficient than multi-threaded software on multi-core processors. Normally they run on multiple processors communicating over a time-triggered bus. Hence the execution is less efficient than it could be. While time-triggered programming model simplifies code generation, we feel that multi-rate event driven execution model is much more efficient. Code synthesis for such execution model must be thoroughly investigated. The multi-threaded software generation is inspired by a recent shift in the hardware design paradigms from single-core to multi-core processors. This shift has brought parallel and concurrent programming to the desktop and embedded arena. In the desk-top market, most processors now being sold are multicore, and very soon this trend might conquer the embedded world as well. Embedded processors like ARM Cortex-A9 or Renesas SH-2A DUAL have already achieved favorable results in implementing multi-core technology.

In this project we have developed formal models, methods, algorithms and techniques for generating provably correct multi-threaded reactive real-time embedded software for mission-critical applications. For scalable modeling of larger embedded software systems, the specification formalism has to be compositional and hierarchical. Our formalism entails a model of computation (MoC) based on a multi-rate synchronous data-flow paradigm. This MoC is code named MRICDF (Multi-rate Instantaneous Channel Connected Data Flow Actors Network) that we developed during two consecutive summer faculty internships at the AFRL in Rome, NY. Once an MRICDF specification is proven to be implementable on a target platform, the corresponding multi-threaded code based on Pthreads, Open-MP, or Intel Thread Building Block can be generated via formal step-wise refinement based algorithms. Our code synthesis is correctness preserving refinement of the original specification into implementation by calculating scheduling that preserves the intent of the specification. Therefore, the generated code does not require expensive post-development testing or verification. Guaranteed determinism of the generated code will provide predictability of the application behavior which is often missing in such complex software created manually or generated from MATLAB/Simulink or Ptolemy like environments. We must also analyze the real-time guarantees that the reactions to specific events should satisfy. The timeliness property is surely platform dependent and hence will require profiling of the code for specific platforms. Back annotations of the specification model with

timing information, and an additional phase of timing analysis will be performed to providing timing guarantees.

In this work we produced a novel theory of a formal modeling language based specifications, namely MRICDF. We demonstrated an implementation of a software specification and code synthesis tool based on MRICDF. This work entails new synthesis algorithms, characterization of specifications, formal proof techniques for proving the correctness preservation property of the refinement steps in our step-wise refinement oriented synthesis technique, multi-core code synthesis, endowing the specification with platform specific worst case execution times to check real-time schedulability, and some case studies.

# Major Highlights

The major highlights of this project are as follows:

1. Two versions of the EmCodeSyn Graphical User interface one based on C++ libraries and another based on C# portable to Linux via Mono. The second version is more robust and scalable.
2. Algorithms for checking if an MRICDF model is synthesizable as a deterministic sequential software, or as a multi-threaded deterministic software, called endochrony check, and weak-endochrony check
3. A completely new semantics and compilation scheme for polychronous models – based on Boolean propositional logic and prime implicate computation
4. Algorithm for multi-threaded code synthesis for weakly endo-chronous MRICDF models
5. New Algorithms for causality detection in MRICDF specifications using SMT Solvers
6. Enhancement of the EmCodeSyn tool to specify task structure with real-time parameters for schedulability analysis
7. Algorithms for real-time schedulability analysis of MRICDF task structured models, and real-time multi-threaded code synthesis
8. A new semantic interpretation of polychrony with conditional partial orders, and use of conditional partial order to do application specific hardware synthesis from Polychrony
9. Combining AADL and MRICDF to allow end-to-end system design for safety-critical systems
10. 3 PhD students funded – one already completed. One postdoctoral scientist trained. Multiple undergraduate research assistants trained. One MS thesis forthcoming.

For keeping the report within reasonable length, we will not describe each of these in detail, but provide the details of a number of selected works. We have published the following papers at international conferences and journals:

1. Bijoy A. Jose, *Formal Model Driven Software Synthesis for Embedded Systems*, PhD Dissertation, August 2011
2. Bijoy A. Jose, Abdoulaye Gamatie, Julien Ouy and Sandeep K. Shukla, *"SMT Based False Causal loop Detection during Code Synthesis from Polychronous Specifications"*,

ACM/IEE 9th Intl. Conf. on Formal Methods and Models for Codesign (MEMOCODE), Cambridge, UK, July, 2011.

3. Jens Brandt, Mike Gemuend, Klaus Schneider, Sandeep Shukla, and Jean-Pierre Talpin, *"Integrating System Descriptions by Clocked Guarded Actions"*, Proceedings of International Forum on Design Languages (FDL'11), September 2011, Oldenburg, Germany. (Invited to Springer Journal of Design Automation of Electronic Systems and current under second review.)
4. A. Matusiewicz, N.V. Murray, and E. Rosenthal. *"Tri-based set operations and selective computation of prime implicates"*. In Proc. International Symposium on Methodologies for Intelligent Systems - ISMIS, Warsaw, Poland, June, 2011, 2011. Lecture Notes in Artificial Intelligence, Springer-Verlag. Vol 6804, 203-213.
5. Jens Brandt, Mike Gemuend, Klaus Schneider, Bijoy A. Jose and Sandeep K. Shukla, *"Causality Analysis of Polychronous Programs*, FERMAT Technical Report 2011-02, 2011.
6. Julien Ouy, Jing Huang and Sandeep Shukla, *"Behavioral Compatibility Checking of Polychronous Components"*, FERMAT Technical Report 2011-03, 2011.
7. Bijoy A. Jose, Abdoulaye Gamatie, Julien Ouy and Sandeep K. Shukla, *"SMT Based False Causal loop Detection during Code Synthesis from Polychronous Specifications"*, FERMAT Technical Report 2011-04, 2011.
8. Jens Brandt, Mike Gemunde, Klaus Schneider, Sandeep K. Shukla and Jean-Pierre Talpin, *"Integrating System Descriptions by Clocked Guarded Actions"*, FERMAT Technical Report 2011-06, 2011.
9. Bijoy A. Jose, Sandeep K. Shukla, *"New Techniques for Sequential Software Synthesis from a Polychronous Data Flow Formalism"*, FERMAT Technical Report 2011-07, 2011.
10. Bijoy A. Jose, Abdoulaye Gamatie, Matthew Kracht and Sandeep K. Shukla, *"Improved False Causal Loop Detection in Polychronous Specificationof Embedded Software"*, FERMAT Technical Report 2011-08, 2011.
11. M. Nanjundappa, M. Kracht, J. Ouy, and S. K. Shukla. Synthesizing embedded software with safety wrappers through polyhedral analysis in a polychronous framework. In ES-Lsyn'12, pages 24 –29, june 2012
12. M. Nanjundappa, M. Kracht, J. Ouy, and S. K. Shukla. A New Multi-Threaded Code Synthesis Methodology and Tool for Correct-by-Construction Synthesis from Polychronous Specifications. In ACSD 2013, Jan 2013
13. M. Anderson and S. shukla, ""APECS: An AADL and Polychrony based embedded computing system design environment with an elevator control case study."," in ACM/IEE International Conference on Formal Methods and Models for Co-Design (MEMOCODE), Portland, 2013
14. "Constructive POLYCHRONOUS Systems". J.-P. Talpin, J. Brandt, M. Gemünde, K. Schneider, and S. Shukla. In Science of Computer Programming. Elsevier, 2014 (to appear)
15. Jens Brandt, Mike Gemunde, Klaus Schneider, Sandeep K. Shukla, Jean-Pierre Talpin: **Embedding Polychrony into Synchrony.** IEEE Trans. Software Eng. 39(7): 917-929 (2013)
16. Julien Ouy, Matthew Kracht, Sandeep K. Shukla: **Abstraction of polychronous dataflow specifications into mode-automata.** ICSAMOS 2013: 33-40

17. Jean-Pierre Talpin, Jens Brandt, Mike Gemünde, Klaus Schneider, Sandeep K. Shukla: **Constructive Polychronous Systems.** LFCS 2013: 335-349
18. Sandeep K. Shukla, Jean-Pierre Talpin: **Guest Editors' Introduction: Special Section on Science of Design for Safety Critical Systems.** IEEE Trans. Computers 60(8): 1057-1058 (2011)
19. Jens Brandt, Mike Gemunde, Klaus Schneider, Sandeep K. Shukla, Jean-Pierre Talpin: **Integrating system descriptions by clocked guarded actions.** FDL 2011: 1-8
20. Prabhat Mishra, Zeljko Zilic, Sandeep K. Shukla: **Guest Editors' Introduction: Multicore SoC Validation with Transaction-Level Models.** IEEE Design & Test of Computers 28(3): 6-9 (2011)

## Disappointments

One of the biggest disappointments was that the majority of last year's funding was withdrawn due to sequestration which hampered the case study phase of the project. We did not get a chance to do a big case study to measure the benefits of code synthesis with respect to hand written code. We started working with the UAV lab at Virginia Tech's Aerospace and Ocean engineering department, but the work was abandoned as the lead post-doctoral researcher Dr. Julien Ouy had to be let go due to lack of funding.

# 2 Introduction

## 2.1 Structuring of the Reported Results

As discussed before, we are going to present a selected assortment of the results obtained during the execution of this project. In Section 3, we describe the MRICDF model, and the corresponding formalism. In Section 4, we report the selected assortment of the results, starting with the combination of AADL formalism with MRICDF, as this is going to be the future work of our group. The next is our work on better check of causality using polyhedral and SMT based analysis, and how to overcome some of the violations. Then we discuss our multi-threaded code synthesis algorithms, and finally the real-time extension of EmCodeSyn.

# 3 Methods, Assumptions and Procedures

This section provides some background information on the methods and techniques that we based our work on.

## 3.1 Programming Model, and Synthesis Technique

In the recent past we have developed a programming model called Multi-Rate Instantaneous Channel Connected Dataflow Actor Model (MRICDF) [1, 2, 3, 4], to capture the specification of a reactive embedded software. We also developed a visual specification and component code synthesis tool called EmCodeSyn [2] which accepts MRICDF specifications in visual or textual form, and produce C-code that is correct-by-construction with respect to its MRICDF specification. Similar to Esterel [18], Lustre [19], and SIGNAL [6], the model of computation of MRICDF is based on synchrony hypothesis [20] which provides a suitable abstraction from computation and communication time, and allows one to focus on the dataflow and computation functionality of the required software. Almost all of these formalisms with the exception of MRICDF are developed in Europe. Airbus [24], Renault, and other European avionics and automotive companies claim to generate a large percentage of their control software using these formal approaches. Even though they have been successful in developing modules through these methodologies, for composition of modules, they use an over simplified composition model based on Time-Triggered Architecture [17]. This requires that each component itself is time triggered, which leads to a number of optimality problems as we pointed out in [1, 4]. The semantics of MRICDF is not time triggered but rather event triggered, leading to more optimal code synthesis [1]. Time triggered composition has another problem other than optimality. It requires precise clock synchronization and the resulting overhead. We want both optimal implementation of individual components, and want to avoid the overhead of time synchronization over a distributed platform. Therefore, constructing large software systems from components synthesized with this tool is much more challenging, but the benefits outweigh the difficulty. In this project we are addressing these challenges.

The programming model of MRICDF is that of a collection of concurrent processes described by data flow relations on infinite streams of data values. The synchronization requirements between these streams are expressed either implicitly by the data flow relations or by explicit constraints. When sequential embedded software is to be synthesized, both data flow relations — computation — and synchronization constraints — control — must be considered. This is the crux of the compilation/synthesis process for MRICDF. This programming model is more suitable for reactive systems compared to other specification models such as temporal logics, composition of automata (such as I/O Automata) etc., because it abstracts away timing issues but most importantly, it makes specification of synchronization between concurrent activities within each component much easier than those other methods. The expression of synchronization between concurrently acting behaviors within a system is a major source of errors (deadlocks, live-locks, violation of mutual exclusion etc.) in other formalisms.

Given a specification (visual or textual) in MRICDF, a compilation algorithm must decide whether there exists deterministic sequential/multi-threaded code satisfying the constraints, and if so, whether it is unique. If not — and thus nondeterministic — the user must provide additional constraints to make it so. If this effort fails, the specification is rejected by the compiler. In the

process of determining implementability, and subsequent synthesis, the compiler creates a *Boolean theory* and computes its prime implicates.

## 3.2 Boolean Theory and Prime Implicates

A Boolean theory is a set of Boolean clauses. Let the theory $B$ be defined over a set of Boolean variables $X$. Then $[X \rightarrow \{0,1\}]$ denote the space of all assignments to variables in $X$.

An assignment $f \in [X \rightarrow \{0,1\}]$ is a model for theory $B$, if and only if by assigning the Boolean values to all variables $x \in X$ as $f(x)$, one can satisfy all clauses in $B$.

A Boolean theory that is satisfiable has at least one such model. A prime implicate of a Boolean theory is a disjunctive Boolean clause $C$ such that any model of $B$ also satisfies $C$ and there is no $C_1$ such that $C_1 \rightarrow C$ and any model of $B$ also satisfies$C_1$.

Given an arbitrary Boolean theory, computing prime implicate is often of exponential complexity. Most previous algorithms also required that the Boolean clauses in the theory be first converted into a conjunctive normal form (CNF) before applying the algorithm. The recent work of Murray and Rosenthal [34] has come up with a new algorithm that can produce prime implicates (and an implicit representation of all prime implicates) of a Boolean theory where the clauses can be in any arbitrary form. However, this algorithm is also time consuming.

We expect that this algorithm can be sped up substantially because the current algorithm is *agnostic* of any special characteristics of the Boolean theory that are generated from MRICDF models during computations of their master triggers.

It has been shown in the past that algorithms that are agnostic of special nature of the inputs on which the algorithm is applied, have higher time and space complexity, than algorithms that take into account special nature of their inputs. For example, finding chromatic number of a graph is known to be NP-Complete, but if we know that the only graphs we need to compute the chromatic number of, belong to a special class of graphs called "Perfect Graphs" then one can come up with special algorithms which can compute the chromatic numbers in polynomial time. Similarly, the famous SAT problem that is a well-known NP-Complete problem can be shown to be solvable in polynomial time, if the clauses we obtain belong to the class of HORN clauses. Also, there is a notion of localization of problem instances. For example, if the variables that occur in multiple clauses can be limited to reappear is no more than k clauses; we say that that SAT problem is k-bounded. In such case, one can devise faster algorithms for solving SAT.

Since the Boolean theories that we generate from MRICDF is very localized, in the sense that a clause $x \leftrightarrow y \vee z$ appears only when $y$ and $z$ are inputs to a merge actor in an MRICDF model, one can find such locality properties. As a result, Murray and Rosenthal's algorithms to compute prime implicates may not exploit such locality (which we referred to as 'regularity') and we may need to devise new algorithms that work much faster to compute prime implicates for such instances.

The prime implicates enable construction of a hierarchical control structure that creates a deterministic schedule of all the computations which is consistent with the control constraints. If non-Boolean constraints — for example, $x > 10$ — are replaced by unrestricted Booleans, the resulting theory is a conservative abstraction of a more elaborate theory with further

expressiveness. The latter would provide better leverage in optimizing the control structure and in reducing redundant paths. To this end, the combination of prime implicate algorithms and SAT Modulo Theory (SMT) solvers [25] is concurrently being investigated.

# 4 Result and Discussion

## 4.1 APECS: An AADL and Polychrony based Embedded Computing Systems Design Environment

### 4.1.1 Introduction

**Overview**: Distributed real-time embedded (DRE) systems are a widely employed in a number of today's safety critical systems in applications ranging from automotive and avionics to medical equipment. These systems are composed of a range of hardware (sensors, actuators, microcontrollers) and driving software. The safe operation of these systems is contingent upon management of correctness of software/hardware resource allocation and adherence to real-time constraints. Verification that the end system behavior and properties adhere to the constraints requires analysis of the system as a whole as well as the individual component properties. The Architecture Analysis and Design Language (AADL) [68] was originally developed for the avionics industry and has since been adapted for use in automotive and other commercial applications. AADL allows for the creation of models that capture the full hardware platform architecture along with the corresponding software hierarchy. Key static properties of the model, such as communication protocols and hardware bindings, are specified as component properties. While software behavior specification is not a part of the core standard, it can be handled by the state-machine like language of the Behavioral Annex, one of the available extensions to the AADL standard. A number of tools have been developed in the past to support the creation of AADL models [69] and for comprehensive system property analysis [70]. Among them is Ocarina [71], a tool for synthesizing executables for a specified hardware platform from an AADL model. With Ocarina, a software component may be associated with a behavioral specification given in a C or Ada source file. More recently, support has been added for specifications from source files generated by Esterel [72] and Lustre [73]. While this is an attractive option, it still presents verification challenges that stem from a lack of formal semantics in the case of the C and Ada or from scheduling and clock constraints when dealing with the synchronous languages Esterel and Lustre. In [74] we discussed introducing the polychronous formalism of MRICDF into Ocarina and its effect on software verification. In this work, we explore the potential advantages of an MRICDF extension when applied to models of multithreaded systems. Ocarina specifies behavior at the function level, associating AADL subprogram calls with a function from a source file. A thread component may be represented by a single such function or it may contain multiples ordered by a call sequence. This requires a manual analysis and decomposition of the software into its thread components. Because the thread synchronization is handled outside of a provably correct synthesis tool, it adds an additional verification obligation to ensure proper synchronization. With synthesized multithreaded code, the addition of synchronization primitives based on preset templates can result in over synchronization if unchecked, possibly creating deadlock situations. Further, synchronous languages like Esterel and Lustre artificially force threads to synchronize at the end of each execution step to respect the global clock constraints. Using MRICDF it is possible to model a software process as a whole, automating the detection and synthesis of its component threads. This has a number of advantages, the first of which is that the synthesis and verification of the thread synchronization is automated and handled internally. Also, because Polychrony

supports a multi-clocked model of execution, threads need not synchronize unless their behavior requires it. In [74], we proposed a new design environment, APECS, for end to end system modeling of DRE systems and automated code synthesis. While in [74] we focused on the environment of APECS, its constituent tools, and the overall approach it takes to modeling, in this work, we discuss the unique advantages and new features it brings to multithreaded code synthesis in the AADL context. Coming sections will provide a brief background of the major tools and formalisms used by this project, as well as an overview of related work in this area, introduce the problems addressed by the new extensions and explain the details the implementation of our synthesis extensions which will be illustrated by a case study. Finally, conclusions and future work will be discussed in last subsection.

**Background and Related Work**

**AADL:** The Society of Automotive Engineers (SAE) created the AADL standard [75]. It is a model based formalism for the comprehensive representation of DREs. AADL utilizes a hierarchical component centric model. Its components can be categorized as either hardware (processor, memory, device, bus), software (process, thread, subprogram, data), and systems. The latter serve as abstractions that represent composites of other subsystems or components. These composite groupings provide the structure that forms model's hierarchy. Data is passed between components through one of a few communication methods. Most commonly, components have some number of I/O ports. These ports are typed and bound to the ports of other components. Alternatively, components may communicate by shared resources, either through buses or by directly accessing shared data. The order in which data passes between ports and through (sub-) components is given in flow specifications. Components may have more than one operating mode; different modes represent different operational states and may have distinct active connections, flows specifications, or operating threads. Each component type may have one or more implementation. Distinct implementations will have the same interfaces, but may have different subcomponents, internal flows and execution modes.

There are a number of development environments available that support the AADL standard. One such is the Open Source AADL Tool Environment, OSATE [69], which is an eclipse based platform. OSATE supports textual representations in AADL and these models persist in XML. It also has a number of extensions that support graphical system modeling. Both the tool and the standard are highly extensible. The AADL standard accepts extensions in the form of language annexes, a number of these extensions already exist with purposes ranging from behavior specification to error modeling. The OSATE development environment supports the addition of plug-ins for analysis and code synthesis. Eventually, we plan to propose MRICDF as a behavioral annex for AADL.

**Ocarina:** One of these tools Ocarina [76] analyzes an aadl model and performs automated code synthesis. The tool's operation can be divided into two distinct modular sections. The frontend contains modules for the AADLv1 and AADLv2 standards. These modules handle the parsing of the input AADL specifications. After lexing and parsing the input, the model is analyzed for syntactic and semantic correctness. From this analysis Ocarina constructs an Abstract Syntax Tree (AST). An Instance Tree is then derived from the AST. The root of the instance tree is the top level system in the model hierarchy. The nodes of the tree are comprised of those component implementations that are subcomponents of the root system. This last step prunes out any

components unused by the current root as well as verifying the presence and accuracy of necessary component properties.

After the frontend finishes building the instance tree, it is fed to the backend. The backend has modules for each of the supported source languages. First it expands the instance tree, simplifying complex structures and annotating them with information for future code mapping. Using the mapping rules for the chosen source module, Ocarina builds an intermediate syntax tree. Finally it traverses this tree and applies the relevant mapping rules to generate source code in the targeted language. This generated code interfaces with the underlying hardware through the PolyORB-HI [77] middleware. The defacto language modules in the Ocarina backend are Ada and C/C++, with an extension that allows it to use the code output from Esterel and Lustre. In [74] we further extended these backend modules to accept code output from MRICDF.

**MRICDF:** Multi-Rate Instantaneous Channel Connected Data Flow (MRICDF [78]) is a formal polychronous dataflow language. Like the other synchronous languages previously employed by Ocarina, it is based on the synchrony hypothesis, an abstraction in which communication and computation are treated as instantaneous. Unlike Esterel and Lustre though, MRICDF's polychronous semantics allow it to pace the activity of the model around the rates of data arrival at individual inputs rather than enforcing synchronization with a global clock.

MRICDF models software as a network of communicating actors. Four primitive actors form the foundation of the MRICDF language. The Buffer actor may be of size N and takes a single input; that input value is stored and reproduced on the output N execution steps later. The Priority Merge and Sampler actors are used to control and route the flow of data between actors. Priority Merge has two input channels, one of which is designated the priority. At each execution step, if it has only one input value, that value is passed through. If it receives two input values, the value on the priority channel is passed through and the other is dropped. Similarly, the Sampler has two input channels, one is an input value and the other is a boolean control signal. When the control signal is false, input values are blocked from passing through to the output channel. The final primitive is the Function actor, (F (n, m)), this actor applies a function F to its n inputs and generates m outputs. More complex behaviors can be modeled and stored as composite actors. Given a completed model specification, MRICDF performs prime implicate based epoch analysis that allows it to formally analyze the models timing as well as detect opportunities for parallel execution [79].

**Related Work:** Another project that attempts to leverage polychrony to formally specify AADL is detailed in [80]. The authors describe a process by which an AADL specification is translated into the Signal language [81]. After being translated, the Polychrony tool is applied; using clock calculus it analyzes the clock relations of the new model and ensures determinism. Similarly, in [82] AADL models are translated into BIP [83] so that they can exploit existing BIP analysis tools. There are a number of other works [84], [85], [86] that deal with the use of formal intermediate representations for AADL for analysis such as model checking and performance evaluation. The Compass tool [87] can be used for fault-tree analysis based model checking of an AADL specification, as well as simulation.

Unlike these approaches, we're not attempting to create a tool for the simulation or validation of AADL models. Instead we wish to create an environment for the development of end-to-end DRE systems and their software behavior. Such a model can be gradually refined and

reconfigured as it is checked for correctness while still providing comprehensive information for checking performance and schedulability. To this end, rather than translating the whole system into an intermediate formalism, we extend Ocarina with support for software from polychronous specifications. These specifications are at the process level, allowing parallelism to be dynamically detected and automatically added to the model before code generation. Information about the process of detecting and exploiting parallelism in MRICDF models can be found in [70].

### 4.1.2 APECS Methodology

**Platform Modelling with AADL:** The first step when building an AADL model is to determine the components that will make up the system. Some components, such as processors or devices, represent tangible elements of the system. Others components, such as the system, can be used as abstractions to organize the system hierarchy. For smaller models it may be possible to start by enumerating all the components made necessary by the specification. However, for larger projects, a top-down approach is the best for organization and expediency. In AADL, the system component serves dual roles. It may be a composite of subcomponents representing a system like a processor board or a software application. Alternatively, it may contain no subcomponents or only other systems, representing a generic system. Such a generic system may be refined later with new sub-system implementations as the design stages progress. So, when creating a top-level model, the first step is to define a system that will encapsulate the complete specification.

Declared as subcomponents within this complete system are the other systems and devices that comprise the specification. In AADL, devices are abstractions used for objects that interact with or are a part of the environment, such as a sensor or motor. A system will be used to encapsulate objects in the system for which it is valuable to enumerate internal composition. It is worth noting that this may include some objects which could otherwise be represented by a device, if there is value in modeling the internal properties of that object. For the sake of organization, each subsystem is defined in a separate package. This allows the subsystems to be included and reused as needed in future.

```
system microcontroller
  Features
    ---
    ---
end microcontroller

system implementation microcontroller.mc
  Subcomponents
    CPU: processor ---
    RAM : memory ---
    Comm_Bus : bus ---
    Main : process ---
  Connections
    ---
end microcontroller.mc
```

**Figure 1: AADL Microcontroller Code**

After enumerating the top level subsystems and devices, the internals of these subsystems must be defined. For complex subsystems, it's necessary to repeat this process of templating with devices and subsystems. Simple subsystems will be modeled with the appropriate hardware and software components. For example, the microcontroller for a board can be modeled through a combination of hardware and software components. The microcontroller itself is made up of processor and memory components communicating via a bus component. Additionally, the driver software is modelled as a process containing at least one thread and bound to the processor it is driving through use of the Actual_Processor_Binding property.

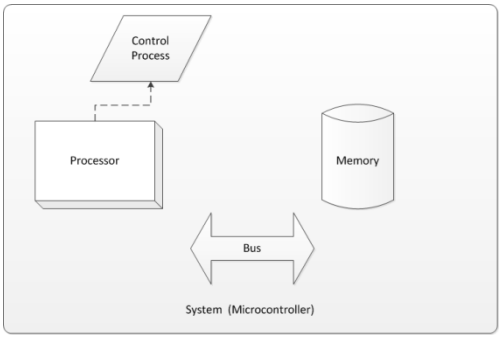



**Figure 2: Microcontroller Layout**

Each component is defined first with a type block. The interface is also defined as part of the type block through the addition of communication ports. Each type block is then associated with one or more implementation block, which contain the specification of subcomponents. The control and data flow of the system are explicitly specified (see Figure 3) in these blocks. The control flows are given by means of end to end flow specifications both between interface ports in the type block and between subcomponents in the implementation block. Data flow is given through making port connections between components.

```
system implementation door.elevdoor
  Subcomponents
    CPU: processor ...
    RAM : memory ...
    Comm_Bus : bus ...
    Main : process ...
    Sensor : device ...
    Motor : device ...

  Connections
    ...
    EC13 : port Sensor.obs -> Main.obs;
    ...
    EC16 : port Main.OpenCmd -> Motor.open;
    ...
  Flows
    ETE : end to end flow Sensor.obs_flow_source -> EC13 -> Main.obs_flow_path -
> EC16 -> Motor.obs_flow_sink {Latency => ...};
end microcontroller.mc
```

**Figure 3 End to End Flow**

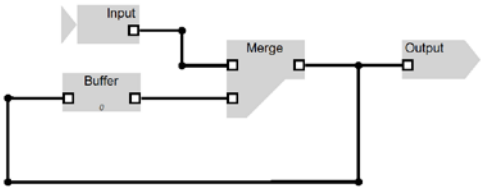



**Figure 4: Latch**

The outputs of the MRICDF software are thus driven by the current state variable values being fed through the combinational actor network.

**Software Modelling with MRICDF**: Once the architecture of the system has been developed in AADL, the next step is to model the software that will drive the controllers. The interface requirements for the software are given by the communication ports featured on the corresponding thread component. A state machine is derived based on the desired behavior specification for the component and the thread interface.

With the desired behavior codified in the state machine, it is time to create the formal software model in MRICDF. The MRICDF model is constructed as an actor network made up of corresponding state variables and combinational data flows. The state variables in MRICDF are modeled as a latch system, comprised of a buffer actor that feeds back into itself through the low priority port of a merge actor (refer to Figure 4). The priority port of the merge is given to some event input, so that state may be updated. The outputs of the MRICDF software are thus driven by the current state variable values being fed through the combinational actor network.

```
thread main
  Features
     ---
     ---
end main

thread implementation main.m
  Properties
       Dispatch_Protocol => ... ;
       Period => ... ;
       Deadline => ... ;
       source_name => "MyMain";
       source_language => MRICDF;
       source_location => "../PATH";
end main.m
```

**Figure 5: Source Code Binding**

**Code Generation:** With the control software for the system formally modeled in MRICDF, it only remains to integrate it into the AADL system hierarchy. The process of associating the two is syntactically straight forward, but giving AADL the capability to exploit this association meaningfully is more challenging. Two string type properties are added to the thread components, specifying the source file name and its location. A third property gives the source file's language. Natively, however, AADL has no capability to interact with the associated programming models. Therefore, in order to support the automated analysis and generation of system code, we propose an extension of the Ocarina [78] tool suite. Ocarina's front end modules

will function just fine, parsing both AADL standards in use by OSATE. However, we will need to create a new backend module so that we can create an intermediate tree with syntax compatible with MRICDF. Further work needs to be done to effectively link MRICDF's executable code with the Ocarina's current middleware.

**Elevator Case Study – AADL Platform Model:** Our case study models a five story building that is being served by four elevators. This means that at the top level the system is composed of the four elevator cars, the button call panels on each floor, and a central controller to handle the scheduling behavior of the elevators.

1) Button Call Panels: The call panels are systems containing a set of devices representing a pair of buttons, a pair of lights, and a sensor that detects when an elevator car is on the floor. There are two additional devices that signify the arrival of an elevator car: a light and a chime. When a button is pressed, its light is toggled on and an event is transmitted to the central controller notifying it of an elevator request call. When the sensor detects an elevator on the floor, two events are triggered. First, the button lights are toggled off. Second, the arrival light is toggled on until the car leaves, and the chime is rung once. All of this is handled in hardware without the aid of a microcontroller.

The call panel is described in its own package. A floor package is then defined. For our model, a floor is a system that contains four call panels and has interface ports for communication between the central controller and the panels.

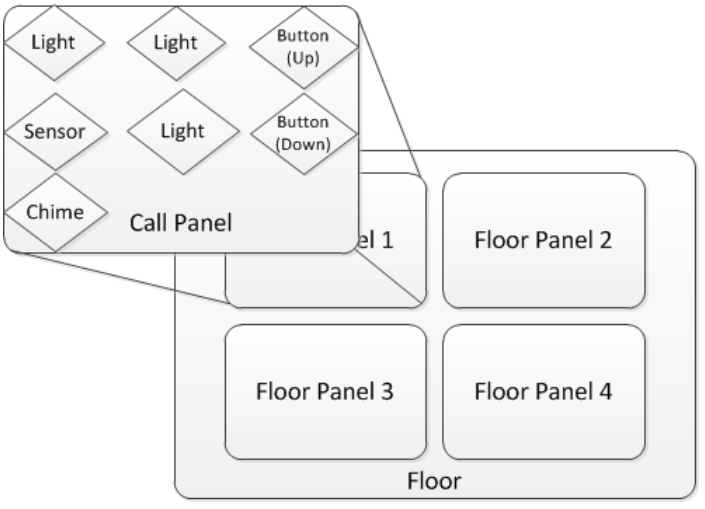



**Figure 6: Floor Model**

2) Elevator Car: The elevator cars are systems that are composed of a number of devices and subsystems. Each car is connected to a device that represent the motor that raises and lowers it in the elevator shaft. The door must be modeled as a system. The door system is composed of a microcontroller, as described in Figure 6, and devices for interacting with the environment: a motor, a sensor, and a timer. The motor opens and shuts the door according to requests from the controller. The timer keeps track of how long the door has been open, and relays that information to the controller. The sensor performs the important safety task of alerting the controller when an obstruction is detected in the path of the door.

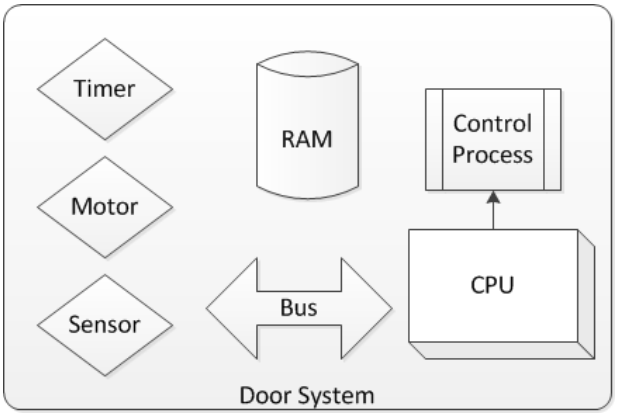



**Figure 7: Door Model**

For the sake of readability in these figures we enumerate only the object composition of the system and not all the communication ports. However, to illustrate the system's device features we provide a sample of the interface for the door control process in Figure 8.

```
process Control features
        openReq : in event port;
        closeReq : in event port;
        isopened : in data port;
        isclosed : in data port;
        obstruction : in data port;
        timeOut : in event port;
        startTimer : out event port;
        openDoor : out event port;
        closeDoor : out event port;
        opened : out event port;
        closed : out event port;
---
end Control;
```

**Figure 8: Control Interface**

Most of these ports are of the event type. For instance, the open and close requests are events triggered by pressing the corresponding buttons of the control panel in the elevator car. The only required information is that the press event occurred, and it is recorded and handled by the internal state of the controller. By contrast, the obstruction signal from the sensor is transmitted over a data port (a Boolean). This is important, because the obstruction may persist for some amount of time and the controller will need to test this value to proceed safely.

Each car also has an internal control panel. Much like the call panels on each floor, this subsystem is composed of numerous button and light devices. There is a button for each floor. There are also two buttons to request that the door open or close and an alarm button. Each button is lit when pressed. Floor selection presses are relayed through the communication ports to the central controller for scheduling, while open and close requests are dealt with internally by the door controller. The alarm presses are relayed outside the system to be handled by an external security service. Finally, there is a microcontroller system driving the car behavior, which will be covered in more detail in the next section. This microcontroller processes information from the button panel and the central controller to relay communications to the door controller and drive the elevator motor. As with the panels, the elevator car is encapsulated in a package for

portability and reuse. For organization purposes, this package is then included in a new system that defines four elevator car objects.

3) Central Control: The central controller is modelled as a microcontroller. It receives information from all the floor systems and elevator bank system and then uses that information to send back scheduling information to the elevator car controllers.

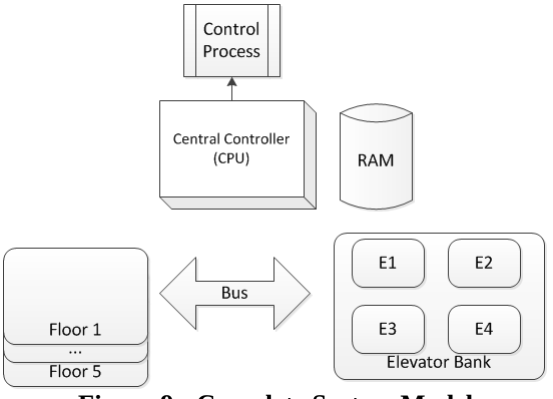



**Figure 9: Complete System Model**

**Behavioral Model:** To ensure correct behavior in an elevator system is complicated by the fact that the behaviors and the controlling models are distributed over a number of communicating, semi-autonomous systems which communicate over buses and specific interface protocols. For example, to arrange a pickup on a particular floor requires:

- The requesting panel communicates with the central controller
- Central Control schedules a pending stop for a car.
- The selected car detects a pending stop on its floor and stops to request that the door opens.
- The door remains open for a minimum set time and until its unobstructed before closing.

In the following sections we will discuss the specialized roles of each controller and its required behavior for desired safety and efficiency.

```
Data: Input: ES1, ES2, ES3, ES4, Request
integer AD1, AD2, AD3, AD4 ;
boolean RW1, RW2, RW3, RW4 ;
while true do
    if Request then
        (reqF loor1, rDir1 ) = extract_request(Request) ;
        (floor1, dir1 ) = extract_direction(ES1 ) ;
        ...;
        (floor4, dir4 ) = extract_direction(ES1 );
        (RW1, ..., RW4 ) = RightWay(dir1, ..., dir4 );
        (AD1, ..., AD4 ) =
         AbsoluteDistance(f loor1, ..., floor4, RW 1, ..., RW 4);

        ChosenElevator =
        findNearestAvailable(AD1, ..., AD4);
    end
end
```

Algorithm 1: Scheduling Algorithm

1) Central Control: The central controllers' job is to process requests from the different floors and schedule elevator cars to perform pick ups. When scheduling the cars, the goal is to send the car best able to reach the target floor quick. To that end:

- The chosen elevator should be currently heading towards the target floor, or it should be idle.
- The chosen elevator should be currently heading in the direction requested by the call button.
- The chosen elevator should be the least number of floors from the target, relative to any other cars meeting the first two criterion.
- Should there be no cars available that meet these criterion, the request should be stored until one becomes available.

It is necessary to wait in the last step, because it is impossible to know which of the busy elevators will become available soonest without knowing what other requests may be made in the interim.

From each call panel, the controller receives up and down call requests as well as whether its elevator car is currently on that floor. From each elevator car, the central controller receives reports of current direction and whether it is ready to move. From these inputs we can extrapolate the values for the central controller states:

- "Pending Request 1-5" This 2-bit state is 0 if none of the panels on that floor have an unserviced request. Otherwise the state is either 10 (Call Up), 01 (Call Down), or 11 (Both). A pending request is cleared when an elevator car has been scheduled for that floor.
- "Elevator State 1-4" Based on the information from sensor panels, these state variables are integers that en- code which floor the elevator is on and in which direction it is headed. The base number is the current floor, which is then filtered based on whether the car is heading up, down, or idle.
- "Ready 1-4" Boolean state that is true if the corresponding elevator car is currently able to move, or false if it is in the processor of picking up passengers.

The states are fed into an algorithm to determine the optimal scheduling. The result of the algorithm is the numeric identifier of the chosen elevator. That elevator will be communicated an event that updates its pendingStop states. Also, at each step, the central controller uses a data port to update the elevator cars on their current position.

2) Elevator Car: The elevator car controller receives as inputs

- "GoTo 1-5" These are input event signals from the central controller assigning stops on their respective floors.
- "FloorNum" The current location of the elevator.
- "FireAlarm" An event indicating the fire alarm has been triggered, puts the elevator into an emergency state until the system resets
- "Floor 1-5" These are input events from the control panel within the car. They represent passenger requests to go to a certain floor and also schedule pending stops.
- "Open" An event input representing a passenger request to open the door.
- "Close" An event input representing a passenger request to close the door.
- "Alarm" An event input representing a passenger re- quest for external aid.
- "Opened" An event signal from the door controller, signaling that the door has opened.

- ”Closed” An event signal from the door controller, signaling that the door has closed.’

The elevator cars travel up and down the building according to the schedule provided by the central controller. A properly behaving elevator will adhere to the following behavior:

- The elevator will continue traveling in the same direction until there are no pending stops remaining in that direction.
- If there are no remaining stops, it will become idle.
- The elevator will not begin to move until the door is closed.
- The elevator will not allow the door to be opened while the car is in motion.
- When the elevator reaches a floor that is one of its pending stops, it will stop moving and request that the door open. If there are other pending stops, it will resume once it is signaled the door has been closed again.

To achieve this behavior, certain internal state variables must be created and tracked from the inputs.

- ”PendingRequests (1-5)” Five Boolean state variables, each track whether a floor has an unserviced request from the control panel and central controller.
- ”NonePending(NP)” A Boolean that is true if all five PendingRequests are false.
- ”LowestPending(LP)” An integer that contains the number corresponding to the lowest floor with a pending request.
- ”HighestPending(HP)” An integer that contains the number corresponding to the highest floor with a pending request.
- ”OnPending(OP)” A Boolean that is true if the current floor has a pending request.

Based on the desired behavior and the internal inputs and states, we can construct a state machine for the elevator car’s behavior.

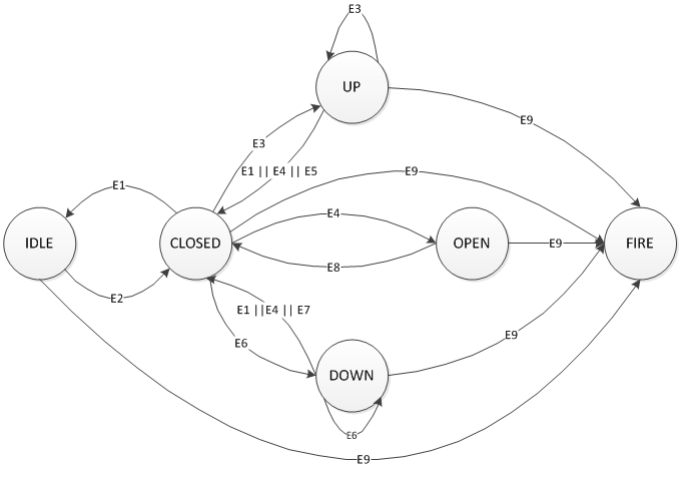



**Figure 10: Elevator Car State Machine**

**Table 1: Event Descriptions**

| Event | Conditions | Outputs |
|---|---|---|
| E1 | NonePending | Move = 0 and Dir = 0 |
| E2 | !NonePending | Move = 0 and Dir = 1 or -1 |
| E3 | (Dir == 1 and HP > CF) | Move = 1 and Dir = 1 |
| E4 | OP | Move = 0 and Dir = 1 |
| E5 | HP < CF | Move = 0 and Dir = -1 |
| E6 | Dir == -1 and LP < CF | Move = -1 and Dir = -1 |
| E7 | LP > CF | Move = 0 and Dir = 1 |
| E8 | Timeout | Move = 0 and Dir = Dir |
| E9 | FireAlarm | *see below |

When a fire alarm is triggered, all pending requests are cleared except for the first floor. All elevators currently in service proceed to the first floor and remain there until the system resets.

3) Door: The door is a subsystem of the elevator car. It receives and processes requests from the elevator to open and close the door. The primary safety concern is that the door should not close on any obstruction. The desired behavior of an elevator door is:

- If an open request is received, signal the motor to open the door and signal the timer to begin a new countdown.
- If an obstruction is detected, signal the motor to open the door and restart the timer.
- If a close request is received, signal the motor to close the door.
- If a timeout from the timer occurs, signal the door to close.
- Notify the elevator of any changes to the status of the door, from open to closed or closed to open.

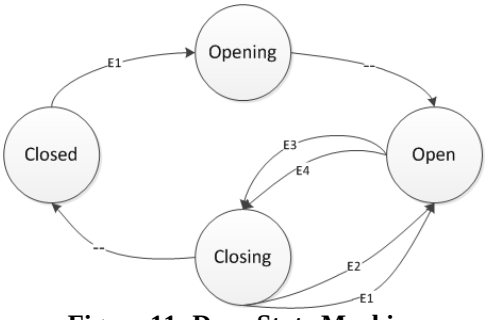



**Figure 11: Door State Machine**

**Table 2: Door Event Descriptions**

| Event | Conditions | Outputs |
|---|---|---|
| E1 | OpenRequest | Motor <= open and Timer <= start |
| E2 | Obstruction | Motor <= open and Timer <= start |
| E3 | CloseRequest | Motor <= close |
| E4 | Timeout | Motor <= close |

In Figure 12 we show the door behavior modeled in MRICDF. The first OR actor will generate a true result.

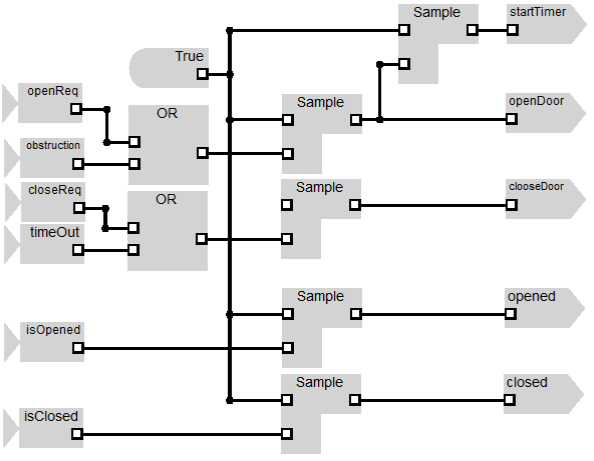



**Figure 12: Door Controller MRICDF**

when an OpenRequest is received or when an Obstruction is detected. A true result from this actor, will activate the sampler to which it is connected, propagating the constant value of true on its input. This event is routed to the door motor and the timer reset. Similarly, the second OR actor receives CloseRequest and Timeout to drive the event sampler for the closeDoor signal of the motor. The lower two samplers are controlled by inputs from the door sensor, and relay events of the door's opening or closing back to the elevator car controller.

The modular nature of this system is ideal for testing different system configurations and iterative design refinement. A hybrid formal software and platform model has advantages over either model alone. The MRICDF models describe the behavior of systems in a manner such that safety specifications are mathematically provable. The platform model, meanwhile, provides concrete information about the capabilities and composition of the underlying system, allowing us to perform more accurate static analyses. In [88] an extension of MRICDF for defining real-time tasks as a layover on actor relations and attributing real-time measures has been developed. Our next step is to integrate that here. However, for now, we can profile the generated code for WCET estimates for schedulability analysis. Using this information and the described code generation techniques we will be able to rapidly create testable binaries for the target platform after each design iteration. Further we can access additional tools and verification suites that aren't a part of the EmCodeSyn environment, such as Cheddar [71] for thread scheduling.

### 4.1.3 Code Synthesis from MRICDF

In this section we'll discuss the methods by which MRICDF code is associated with an AADL process (Shown in figure 13) and how multithreading has been implemented. Past Ocarina implementations have manually associated each subprogram with a specific function through the use of specialized properties. Similarly, we use properties to associate MRICDF specification with a particular process, as illustrated in figure 14.

```
process controller
        Features
                ---
                ---
end  controller

process  implementation controller.c
        Properties
                source_name  =>  "MRICDF_Source";
                 source_language  =>  MRICDF;
                 source_location  =>  "../PATH";
end  controller.c
```

**Figure 13: Source Code Binding**

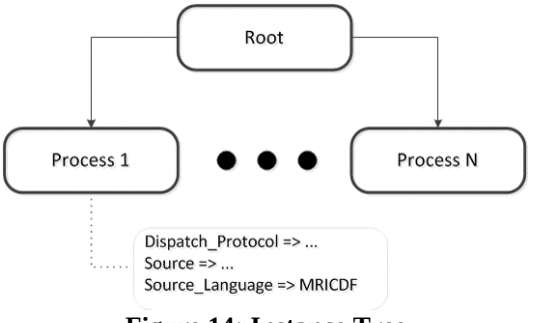



**Figure 14: Instance Tree**

These properties provide the name and location for Ocarina to find the MRICDF network specifying the intended behavior of this process. The source language tells Ocarina which backend module to employ. After the Ocarina frontend has parsed the AADL files, it builds an instance tree containing a hierarchical representation with the top level system as the root and all of the component implementations currently being used by that system as child nodes.

However, unlike other Ocarina sources, this instance tree doesn't yet have any thread subcomponents associated with its processes. The logical next step then is to populate the tree by appending thread instances to the tree. This is accomplished by first analyzing the source file and determining how many threads exist within the process. For each, a thread instance is added as a child node of the process. The thread is then annotated with its critical properties as shown in figure 15.

```
thread <name>
        features
                ...
end <name>;

thread implementation ...
        subcomponents
                ... : data ...;

calls {
        ... : subprogram ...;
};

properties
        Dispatch_Protocol => ...; Period => ...;
end <name>.<subname>;
```

**Figure 15: Generated Thread**

Data exchange for thread input and output is accomplished by accessing shared data components. Property entries are added to specify the dispatch protocol and timing requirements of the thread. Finally, the behavior for each function is specified by adding a single subprogram call. MRICDF generates N function threads for the process, named ”Block1” through ”BlockN”, each subprogram is associated with its corresponding block function (shown in figure 16).

```
subprogram <name>
features
...
Properties
        source_language => C;
        source_name => "block#";
         source_text => ("blocks.cpp");
end <name>;
```

**Figure 16: Associating Thread Behavior**

**Elevator System Code Generation**: To better illustrate these extensions, we’ll use an example of an Elevator System [74]. For the purposes of this example we’ll consider a system of four elevator cars, servicing a five story building. The scheduling of these elevators is handled by a central controller that processes the current state of each elevator along with the requests from the call panels on each floor. Figure 17 gives an overview of the top level system and its subcomponents and system for the AADL elevator model.

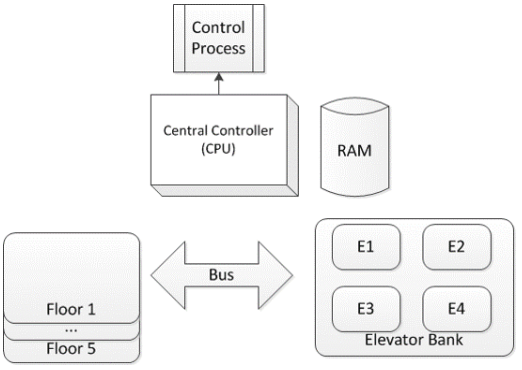
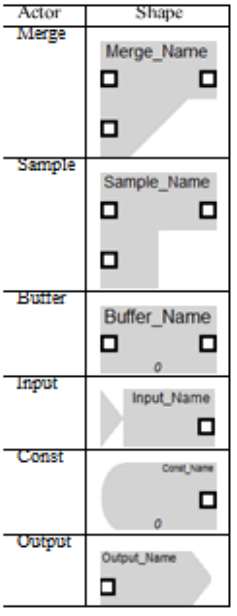



**Figure 17: Top-Level System**

The full details of the system are outside the scope of this work, more information can be found in [74], but some of the subsystems are illustrative of MRICDF's multithreading potential.

**Table 3: Each Actor and Corresponding Shape**




Once the specification is of the door controller (see fig 12) is complete, MRICDF applies clock epoch analysis [78] to examine its clock relations and look for potential parallelism. Figure 18 shows the clock tree extracted from the model.

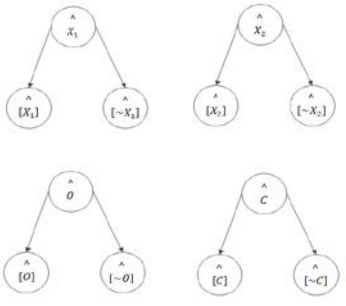


**Figure 18: Clock Trees**

In this case, we have a forest of clocks rather than a single tree. This is because there are no shared dependencies between the output operations. As a result of this, MRICDF creates an independently functioning thread for each tree. One for opening the door, one to close it, and two that update the elevator car controller on the status of the door.

**Another Example**: The door control process provides a succinct example of how our extensions can exploit opportunities for parallelism, but it happens that all of the threads in that process are wholly independent from one another. Now, we'll look at an example where the threads must synchronize with each other occasionally. Consider an alternative implementation of the elevator door, to conserve resources we implement the control so that it only checks passenger requests periodically rather than constantly. This is a reasonable assertion, because the system will be operating much faster than human input and the requests aren't safety critical (unlike obstruction detection)

```
input: event SD, obs;  bool open, close;

[(cnt mod 10) = 0] ^= [rcnt == 5]
open ^= close

rcnt := rcnt $ init 0 + 1 when req default
        0 when (req = zreq |
zreq := req $ init 0 |
req := 1 when open = true default
       0 when close = true default
       zreq |
zcnt := cnt $ init 0 |
cnt := znt + 1 when SD default
       59 - zcnt when CD default
       zcnt |
dir := 1 when obs default
       1 when (req = 1) default
       -1 when (req = 0) default
       0 when cnt = 60 |
zdir := dir $ init 0 |
ChangeDir := true when dir = zdir |
```

**Figure 19: Periodic Controller**

The polychronous pseudo code for these requirements is shown in Figure 19. The variable req reads the current request values out of shared memory. Each time a different request arrives the request counter, rcnt, is reset until a value has been constant for five execution steps. Meanwhile, cnt is tracking the progress of the motor as it opens the door, incrementing on each six degree input event. Direction is tracked by dir which updates based on requests, but only after a request has been stable for five execution steps. If a change in direction is detected, the count is inverted and the process continues. The defined clock relationship between cnt and rcnt, [(cntmod10) = 0] ^= [rcnt = 5], serves as a synchronization point between two partially ordered reaction sets. Updating reqs is totally ordered, with the clocks open ^= close ^= req. Similarly, cnts updates are totally ordered, with clocks SD ^= zcnt ^= obs^= cnt. However, the timing of the interactions between these reaction sets isn't fully known, only that the occurence of [rcnt == 5] coincides with [cnt mod 10 = 0]. Thus MRICDF will create a barrier synchronization at that point, whereupon data may be exchanged and dir gets updated. Thus, in the final code synthesis, two dependent threads are generated.

**Conclusions and Future Works**: In this work we present the APECS development environment for as a tool suite for the modeling of DRE systems. By leveraging MRICDF's software specification and verification capabilities with the AADL standard's full hierarchical models, APECS is can be used for end to end development of such systems, including analysis of software behavior and code synthesis for the final targeted platform. The result is a flexible environment capable of iterative model refinement and analysis by an extensible tool suite. We utilize the polychronous clock analysis techniques of MRICDF to detect opportunities for parallel execution and to then dynamically generate the corresponding threads.

In the future we plan to extend this work with further improvements to thread timing and scheduling analysis. One such consideration is adding the option of annotating the generated threads with new, individual timing requirements. This will be further improved upon by incorporating developments from work that's been done on real-time MRICDF [88]. Generating threads for handling bus communication between software and other devices and processes is also a priority step towards expanding the analysis incorporated the software behavioral interactions occurring across the distributed platforms.

## 4.2 A New Multi-Threaded Code Synthesis Methodology and Tool for Correct-by-Construction Synthesis from Polychronous Specifications

Embedded software systems respond to multiple events coming from various sources – some of which are temporally regular (ex: periodic sampling of continuous time signals) and some are intermittent (ex: interrupts, exception events etc.). Timely response to such events while executing complex computation, might require multi-threaded implementation – threads responsible to compute reactions, threads responsible for Input/Output of regular events, and threads dedicated to intermittent events. Multi-core embedded processors are also becoming common in the market. As a result of these, design of multi-threaded embedded software is gaining increasing importance. However, manual programming of multi-threaded programs is error prone, and proving correctness is expensive. In order to guarantee safety of such implementation, we believe that a correct-by-construction synthesis of multi-threaded software from formal specification is needed. It is also imperative that the multiple threads are capable of making progress asynchronous to each other, only synchronizing when shared data is involved or information requires to be passed from one thread to other. Especially on a multi-core platform, lesser the synchronization between threads, better will be the performance. Also, the ability of the threads to make asynchronous progress, rather than barrier synchronize too often, would allow better real-time schedulability.

In this work, we describe our technique for multi-threaded code synthesis from a variant of the polychronous programming language SIGNAL, namely MRICDF, and through a series of experimental benchmarks show the efficacy of the tool we developed based on our synthesis technique. Our tool EMCODESYN which was built originally for sequential code synthesis from MRICDF models has been now extended with multi-threaded code synthesis capability. Our technique first checks the concurrent implementability of the given MRICDF model. For implementable models, we further compute the execution schedule and generate multi-threaded code with appropriate synchronization constructs so that the behavior of the implementation is *latency equivalent* to the original MRICDF model.

### 4.2.1 Introduction

Consider a cruise control system of a car that is implemented based on a proportional integral (PI), with *vr* as the target cruise speed, *v* as the actual sampled speed, *T* as the number of samples between two subsequent control thrust (actuation) outputs *u*. The pseudo-C code for this system is shown in the Listing 1, where *S* is the local variable accumulating the integral and *ki* are *k* are constants determined based on PI control. In this pseudo-C code, *Sample(v)*, and *Output(u)* are input/output actions. Now, consider the control loop for the temperature (AC) control system in the same car. Assuming the same PI control paradigm, the pseudo-C code for that is shown in the Listing 2, where *S* is integration summand and *ci* and *c* are constants.

| Listing 1. Cruise Control System | Listing 2. AC Control System |
|---|---|
| L : S = 0 ;<br>Thrust_Interval = T ;<br>while(Thrust_Interval != 0){<br>  Sample v ;<br>  e = vr - v ;<br>  S = S + e * ki ;<br>  Thrust_Interval = Thurst_Interval - 1 ;<br>}<br>Sample v ;<br>u = k * (vr - v ) + S ;<br>Output(u) ;<br>GOTO L ; | L : S = 0 ;<br>AC_Interval<br>while(AC_Interval != 0){<br>  Sample p<br>  e = pr - p ;<br>  S = S + e * ci ;<br>  AC_Interval = AC_Interval - 1<br>}<br>Sample p ;<br>w = c * (pr<br>Output(w) ;<br>GOTO L ; |

In the AC control system, *p* denotes the currently sampled temperature, *pr* denotes the target temperature set by the thermostat, *w* is the signal which controls actuators to release hot air or cool air, and speed of air. Note that, the AC control loop and the cruise control loop might be working at different sampling rates, and their actuation intervals (*Thrust_Interval* and *AC_Interval*) could also be different. If both these control loops are run on the same processor, and scheduled using a real-time scheduling algorithm (with *T* and *T'* being the respective deadlines, and periods for the two tasks), one could easily implement them as two real-time processes. As these two processes do not have any interaction, there is no dependency or no need of any synchronization, and in that case the job of the embedded software designer is simple. Now, consider the possibility where, as the sampled temperature goes below a certain threshold, the cruise control is to be disengaged to manual control, because such low temperature might be indicative of icy weather conditions. This is not necessarily an ideal automotive design example, but rather concocted to make a point regarding multi-threaded control. If the temperature loop is tasked to generate an interrupt and the interrupt is input to the cruise control loop to disengage it, then we have two processes or threads which interact, and timely response to the interrupt needs to be guaranteed. The pseudo-C code for both control systems with interrupts is shown in Listing 3 and 4. In this code, we assume that the *intrpt* is the name of a single bit buffer, whose value is set to *true* or *false*, depending on if the temperature control wants to send interrupt or not. Since this is shared buffer, a semaphore mechanism is assumed to synchronize the read/write of this buffer. The semaphore effectively enforces a barrier synchronization between the two control threads at their outer loops.

This is too simple an example, and hence, getting this synchronization correct is trivial with the use of a 1 bit semaphore. However, in general, multiple threads may need to synchronize at various places of their execution with different threads, and overall behavior must be deterministic. Guaranteeing determinism with multiple synchronizations among a group of threads, while also ensuring no deadlock is often hard and error prone.

| Listing 3. Cruise Control System with Interrupts | Listing 4. AC Control System with Interrupts |
|---|---|
| L : S = 0 ;<br>Thrust_Interval = T ;<br>while(Thrust_Interval != 0){<br>Sample v ;<br>e = vr - v ;<br>S = S + e * ki ;<br>Thrust_Interval =<br>Thurst_Interval - 1 ;<br>}<br><br>P(semaphore2) ;<br>Read intrpt ;<br>V(semaphore1) ;<br>if(intrpt) GOTO MANUAL_MODE;<br>Sample v ;<br>u = k * (vr - v ) + S ;<br>Output(u) ;<br>GOTO L ; | L : S = 0 ;<br>AC_Interval = T';<br>while(AC_Interval != 0){<br>Sample p ;<br>e = pr - p ;<br>S = S + e * ci ;<br>AC_Interval =<br>AC_Interval - 1 ;<br>}<br><br>Sample p ;<br>w = c * (pr - p ) + S ;<br>P(semaphore1) ;<br>intrpt = p<0? TRUE:FALSE ;<br>V(semaphore2) ;<br>Output(w) ;<br>GOTO L ; |

We want that, the threads responsible for distinct control functions must make progress asynchronous to each other except when they interact. Also, we must make sure that the threads are not over synchronized. Thus, in this example, the synchronization is done at the outer loop of the control and not the sampling loop, which would make the sampling rates in the two threads dependent on each other and slow down progress in making corrective actions. We also want to make it easy for designers to decide which variables are to be shared (in this case the *intrpt*), and ensure that when a new value is written, it is eventually read by the other thread, and that it does not read the same value twice. We also, do not want that the absence of an interrupt hold up the other thread too long, and hence absence is encoded as *false*. Such decisions can be taken by the programmer while programming in C or other programming languages, but then proving correctness (i.e. to prove that synchronization indeed guarantees that every interrupt is responded to, and absence of interrupt does not hamper progress, and that there is no deadlock) is much more involved – especially when the number of threads and number of synchronization points are large. If we can capture these requirements in a simple formal model, write appropriate constraints, and generate multi-threaded C-code with appropriate synchronization code, and this code-generation is provably correct, the validation overhead will be much reduced.

In synchronous programming languages such as Esterel, Quartz or Lustre, these two loops will be modeled as two distinct processes since they do not need to move forward by synchronizing at every macrostep. If the two threads were modeled in standard Esterel, it has to be over designed by making two parallel synchronous threads that synchronize quite tightly. To achieve the independence of the sampling rates and the thrust generation intervals by the two loops, one has to model them as separate independent processes. Therefore, the proof of determinacy of interaction for these have to be reasoned at a meta-level. The interaction between the two processes will be external to the model of the processes. Thus proving the correctness

will also be done outside the code-synthesis step. Since our goal is 'correct-by-construction' code synthesis, ideally, the code-synthesis step should guarantee 'correctness' without external reasoning about the generated code. Therefore, we have chosen polychronous modeling paradigms such as SIGNAL or MRICDF, and developed techniques for multi-threaded code synthesis which includes the correct synchronization between two asynchronously progressing threads, with synchronization on a need basis, and guarantee determinacy.

In order to model these two control loops in MRICDF or SIGNAL, we first create two subprocesses, with the cruise control subprocess having an extra Boolean interrupt as a shared variable *intrpt*. When there is a temperature constraint violation, the second subprocess will set this shared interrupt variable to *true*, otherwise it will set the value to *false*. All we have to specify is that the two subprocesses synchronize on reading and writing of this variable, and rest is taken care of code generation. The code generation first needs to prove that the synchronization can be done deterministically, and without possibility of deadlock. Only then it will progress to generate code, by adding synchronization primitives.

This entire process is done by a simple clock calculus on these processes. In the first process, the clocks of *v*, *S*, *e* are the same, whereas the clock of *intrpt* is a subclock, which is the same clock as that of the thrust output *u*. In the second process *p*, *S*, *e* have the same clock, which is possibly distinct from the main clock of the other process because the temperature sampling may be less frequently done than speed sampling. However, an additional constraint in the MRICDF or SIGNAL model will be provided in the model which says that the clock of *intrpt* must synchronize. This statement states that the AC control process must rendezvous with the cruise-control process when it is time to read/write the interrupt. Thus, the two processes can be implemented with two separate threads, which will only synchronize on an interrupt. This multi-threaded process will be deterministic – that is, for the same flow of input events on the sampled speed and temperatures, same flow of outputs will occur. One could make other design decisions such as not reading or writing interrupt that often, so they could consider creating further conditions for read/write of the interrupts. However, for the correct synchronization synthesis to work, both the processes must independently be able to compute such condition. For example, if they are supposed to exchange interrupt information every *n* times thrust is generated, and every *m* times temperature control actuation is generated, that is easy to express as well.

1) *Novelty in Our Approach*:

As we have argued, the polychronous (multi-clock) nature of SIGNAL/MRICDF, will allow the system to be modeled as a single process and yet yield for 'correct-by-construction' multi-threaded code generation. Also, the reasoning about determinism can be done on the whole system, without the need for making any assumptions on the occurrence of interrupts, as the reasoning will be embedded in the polychronous clock calculus.

In our work, we have chosen MRICDF - a multi-rate data flow language, as the formal modeling language. It is akin to SIGNAL, but graphical and added advantages such as rich data types, predefined blocks, etc. A graphical tool, EMCODESYN[4] analyzes the MRICDF models and checks for implementability before generating code. The existing tool checks only for sequential implementability by conducting, a static *Epoch Analysis* (explained later) on a set of Boolean equations derived from the MRICDF model. This analysis is based on the Boolean theory and prime implicates [5]. The challenge we address in this work is – automatic generation

of multi-threaded code that behaves deterministically. We extend the capabilities of EMCODESYN tool, with a novel technique for checking the concurrent implementability of MRICDF models. In this work, we particularly focus on efficiency of the generated code and the practicality of the proposed approach. The proposed technique involves identification of systems that are weakly-endochronous and if found implementable, the technique further generates the execution schedule and multi-threaded code with appropriate synchronization constraints that conforms to the schedule. We have implemented these in EMCODESYN tool and conducted experiments to test performance and scalability issues. It should be noted that a similar idea could be used for generating multi-threaded code for systems specified using SIGNAL language as well. In fact, the theory of weakly hierarchical processes developed in [12] for SIGNAL forms the basis of our work. However, other approaches to SIGNAL multi-threaded code generation are quite different. The approach in [10] approaches the problem with extreme fine granularity by enumerating all possible reactions and computing dependence, and the resulting complexity of the synthesis is very high. On the other hand, such fine grain approach attempts to exploit all possible concurrency whereas we focus on identifying threads that are rooted at distinct incomparable clocks in the clock hierarchy.

Contributions:

1) A novel technique for determining concurrent implementability of the MRICDF models based on prime implicate theory
2) Technique for generating execution schedule and multi-threaded code with appropriate synchronization constraints for implementable models
3) Experimental results showing the scalability of the proposed technique and comparing efficiency of the generated code as compared to hand written code

## 4.2.2 Definitions and Overview of Concepts

A Multi-Rate Instantaneous Channel-connected Data Flow (MRICDF) model is a data flow network model that consists of several synchronous modules called as *actors*, that are interconnected using *channels*. An actor represents a computation with an input interface and output interface for input and output signals respectively. Actors communicate with each other via channels using *signals*. Communication is instantaneous and channels can have different communicating rates. In all, a MRICDF model represents a network of synchronous modules with multiple clocks, which is the basic definition of a polychronous system.

In the polychrony model of computation, *events* form the primitive entities. An *event* is said to have occurred whenever there is a change in the value at an input or output port, or change in value of a variable etc.

Definition 1: **(Event)** We use $\Xi$, to denote the set of all events, and ,$\leq$ to denote a preorder relation among events which indicates the precedence of one event over another is a preorder on $\Xi$: e $\leq$ f means that, event e occurs before or concurrently with event f. ~ is the equivalence relation based on $\leq$: e ~ f means that, events e and f occur simultaneously, also called as synchronous events.

A logical instant, could be thought of as a maximal set of computations that occur in reaction

to one or more events. This set of computations is maximal in the sense that, any other activity would require another value to arrive on those inputs which triggered the current set of computations. Events within one logical instance are all synchronous with each other.

Definition 2: **(Logical Instant or Instant)** We use $\Upsilon$ to denote the quotient of $\Xi/\sim$, the set of logical instants. Thus a logical instant is a maximal set of events that are synchronous.

The synchronous events within a logical instant may be bound by data dependencies and hence are also ordered by a relation. All the dependency relations are captured in the data dependency graph. This relation is not defined as an order but the implementation of the specification is only possible if the dependency relations do not form a cycle, since it induces an order of computation during code generation phase.

Definition 3: **(Signal, Epoch, Clock and Clock tree)** Let $T$ be the type representing set of values a signal can take, $\perp$ be a special value used to denote the absence of the signal, and $T_\perp = T \cup \{\perp\}$, then we can define a signal as a function $\Upsilon \rightarrow T_\perp$.

For a given signal $x$, there exists one maximum set of instants $\gamma \subset \Upsilon$, such that is a total order in and the signal $x$ takes a value from $T$ in each of the instants of $\Upsilon$. Such a set is called the epoch of the signal represented by $\sigma(x)$.

The clock of a signal is a characteristic function that tells if a signal $x$ is present or absent at any given instant $t$ in $\Upsilon$. Clock is a function $(\Upsilon \rightarrow T_\perp) \rightarrow \Upsilon \rightarrow \{true, \perp\}$ that for a signal $x$ returns another signal ($\hat{x}$) defined by: $\hat{x}(t)$ = true if $x(t) \in T$ and $\hat{x}(t) = \perp$ if $x(t) = \perp$.

A signal is a stream of values that occur at specific instants. The epoch of a signal is a set of all logical instants at which the signal is computed or assigned new values. The clock of a signal is a boolean signal that tells the presence or absence of the signal. Not all signals at the interface are present and computed or assigned input values at every logical instant. Thus signals may have different clocks – hence the model of computation is called polychronous or "multi-clocked".

Using the clock relations, a hierarchy of clocks can be built and the resulting hierarchical structure is a clock tree or a forest of clock trees depending on whether the hierarchical structure is single rooted or multi-rooted.

Based on the above definition, signals can be classified into,

- signals $x$ and $y$ are synchronous to each other if their clocks are same: $\hat{x} = \hat{y}$.
- if signal $x$ has events in a subset of instants where signal $y$ has events, then $\hat{x}$ is a sub-clock of $\hat{y}$.
- if signal $x$ and $y$ do not have events that belong to same logical instant, then their clocks can be either mutually exclusive or they are unrelated.

The information regarding clocks of all signals is stored in clock tree.

Definition 4: **(Data Dependency)** We use $\rightharpoonup$ to express data dependency between events. The binary relation e $\rightharpoonup$ f means, e has to be computed after f, in other words, f precedes e.

If the relation $\rightharpoonup$ holds between some pair of synchronous events of two signals, then the data dependency is elevated between those signals. $\forall$ signals $x$, $y$ and $c$, $\forall t\ \sigma(c)(t),\ x(t) \rightharpoonup y(t) \Rightarrow x \rightarrow^{c} y$. We can also write $x \rightarrow y$ for the approximation of $x \rightarrow^{c} y$ on any clock.

Definition 5: **(MRICDF Actors)** Actors in MRICDF language can be classified into two groups,

(a) Primitive actors
(b) Composite Actors. The four primitive actors are,

- **Function Actor**: This actor performs any user specified computation in any instant when the inputs have an event. All the inputs and outputs are synchronized with each other.

Operation: $r = a * b$
Clock relation: $\hat{r} = \hat{a} = \hat{b}$
Boolean relation: $b_r = b_a = b_b$
Dependency relation: $r \rightarrow a,\ r \rightarrow b$

- **Buffer Actor**: This actor is used to temporarily store a value of a signal across instants, in other words – it delays a signal. The signal must have events in both storing and retrieving instants. Increasing the buffer size of the Buffer actor produces the same effect as a series of unit sized Buffer actors cascaded. Both input and output are synchronized with each other.

Operation: $r = b\ \$\ n\ init\ v_1 \ldots v_n$
Boolean relation: $b_r = b_b$
Clock relation: $\hat{r} = \hat{b}$

- **Sampler Actor**: This actor is used to down-sample a signal based on a known Boolean condition. This actor produces outputs in all instants where there is an input and the Boolean condition evaluates to true. Hence the output clock is the intersection of input clock and the clock when Boolean condition is true.

Operation: $r = a\ when\ b$
Clock relation: $\hat{r} = \hat{a} * [b]$, where [$b$] = b is *true*
Boolean relation: $b_r = b_a\ and\ b_{[b]},\ b_b = b_{[b]}\ or\ b_{[\bar{b}]},\ b_{[b]}\ and\ b_{[\bar{b}]} = false$
Dependency relation: $r \rightarrow a$

- **Merge Actor**: This actor merges two signals (can have different clocks) with a higher priority for one of the signal. The clock of the output signal is the union of the clocks of the participating input signals.

Operation: $r = a\ default\ b$
Clock relation: $\hat{r} = \hat{a} + \hat{b}$
Boolean relation: $b_r = b_a\ or\ b_b$
Dependency relation: $r \rightarrow a,\ r \rightarrow b$

Composite actors are hierarchical combination of several primitive actors.

**Master Trigger and Sequential Implementability**

Given an MRICDF model, we need to translate it to runnable reactive software. However, before translation one has to ensure if the given model is actually implementable in software or not, in other words – one has to determine the sequential implementability of the model. To do so, we have to identify the mapping from the abstracted MRICDF entities to actual software. Out of the many mappings, we discuss two important ones here. First one is the mapping of the discrete time representation of the synchronous specifications onto the real continuous time. To achieve this, we have to identify a set of instants that are totally ordered to get a deterministic execution. Since time is a continuous quantity, to map each instant to a time interval, we need a reference signal that is present in each of the instants. In our terminology, this special signal is called as –*Master Trigger*. The existence of *master trigger* is a necessary condition for sequential implementability. To identify a signal which can be *master trigger*, we perform epoch analysis on the given model. In this process, we first construct a Boolean formula, which is the conjunction of Boolean relations for all the actors in the given model. Identifying *master trigger* for the MRICDF model is equivalent to identifying *prime implicate* in the constructed Boolean formula. We use a SMT based technique to identify the prime implicate.

The second important mapping is the scheduling of the computations within an instant. The order of computations is constrained by the data dependencies implied from the specifications. These specifications are represented in the dependency graph and can be easily read while determining the order of computations.

After identifying the *master trigger*, we can generate schedule of computations using clock tree and dependency graph, and eventually we generate the sequential code from this schedule. Formal description of the schedule and code generation technique can be found in [8].

## 4.2.3 Concurrent Implementability

Consider a simple MRICDF model shown in Figure 20(a). For readability purposes, its textual representation is shown in Listing 5. It has 2 input signals *v* and *u* and 1 output signal *w*. Internal signal *x's* value is computed in instants where *v* has events. Similarly *y* is computed in every instant where *u* has events, but *w* is computed in instants only when x >= 10, y >= 20 and both *u* and *v* have events. Hence we can say that, $x \sim v$ and $y \sim u$ as $\hat{x} = \hat{v}$ and $\hat{y} = \hat{u}$. Clock tree for this model is shown in Figure 20(b), where each node represents a unique clock. The labels on the arrows indicate the constraints from which one clock can be derived from the other. Nodes with multiple incoming edges, represent the clocks which can be derived only when the constraints on all the incoming edges are satisfied. It must be noted that the clock tree is not single rooted. This means that there is no single signal which could be used as a *master trigger*. If our aim was to generate sequential code, then we can synthesize a temporary signal that has events when either *u* or *v* or both have events, and then use this temporary signal as *master trigger*. With the addition of this temporary signal to the clock tree, it becomes single rooted. But here, our aim is to synthesize multi-threaded code and hence, we are not concerned with making the clock tree single rooted. For multi-threaded code, instead of a single *master trigger*, we have a set of *partial triggers*. Each of these *partial triggers* act as *master trigger* for the sub-processes. There can be instants where some of the partial triggers are present and some are absent but there cannot be

any instant which all the partial triggers are absent. Clock tree for such a system will have multiple roots as shown in Figure 20(b).

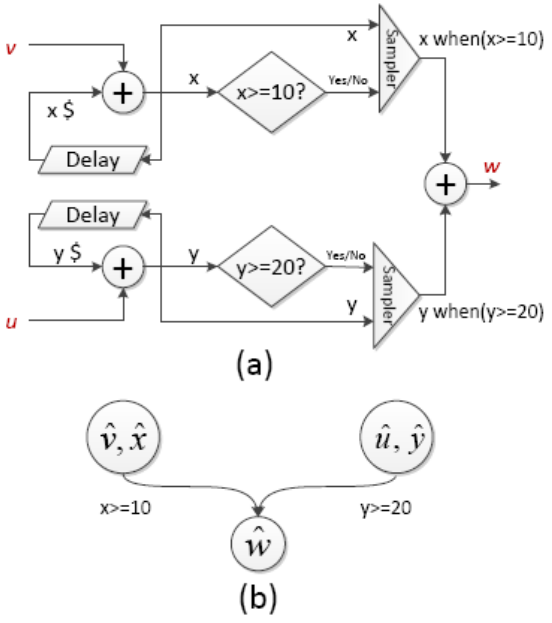



**Figure 20 (a) MRICDF model, (b) simplified clock tree**

Listing 5. Textual Representation for MRICDF model in Fig. 1

```
1  process P = (? integer u,v; ! integer w)
2  (| x := x$ init 0 + v
3   | y := y$ init 0 + u
4   | w ^= (x >= 10) ^= (y >= 20)
5   | w := x + y when w
6   |)
```

Definition 6: **(Partial Triggers)** Let *P* be a MRICDF model representing a data flow process and let *y* be any signal in the process. A set of signals $S = \{x_{1...}x_n\}$ belonging to *P* , is considered as a set of partial triggers iff,

- $\forall y \in P, \exists x_i \in S$, such that y is absent $\Rightarrow x_i$ is absent, i.e, each signal which is not a partial trigger has to have an epoch that is subset of epoch of some partial trigger.

- $\nexists x_1, x_2 \in S$, such that $x_1$ is absent $\Rightarrow x_2$ is absent, i.e, no two partial triggers are under the same clock tree because each sub-process can have only one master trigger

The partial trigger set S is minimal.

### A. Constraints for Concurrent Implementability

In [12], the authors define a class of processes for which concurrent scheduling is

deterministic. This class consists of processes composed of individual sub-processes with their own triggers. A list of conditions that identify those processes was also proposed. Let $P$ , be a MRICDF model representing a data flow process that consists of numerous sub-processes. P can be scheduled concurrently if,

1) The process P can be partitioned into multiple sub-processes $\{P_1,\ldots P_n\}$ and $\{x_1,\ldots x_n\}$ represent their respective master triggers.
2) The dependency graph of the process $P$ does not have cycles.
3) $P$ is well-clocked: the relations between epochs inside subprocesses are compatible at the level of the process. In other words, scheduling of sub-processes does not result in a deadlock.

Considering those rules, we define the criteria for concurrent implementability as follows–

- For each signal $y \in P$, there exists at least one partial trigger $x \in \{x_1,\ldots x_n\}$, such that epoch of y is a subset of and can be derived from epoch of partial trigger x, i.e, $\forall y \in P$, $\exists x$ such that, $\sigma(x) \supset \sigma(y)$ and $\exists f$ a Boolean function such that for each t of $\sigma(x)$, $\hat{y}(t) = f(\hat{y}(t))$.
- Cyclic causal loops are identified by traversing the dependency graph and evaluating the dependencies [9].
- If a process $P$ has $n$ sub-processes, then the clock trees of the sub-processes intersect at most $n-1$ times.

This intersection is due to computation of some sub-process being dependent on computation of other sub-processes. We represent intersection of two processes $P_i$ and $P_j$ as $P_{i,j}$, such that $P_{i,j} \subset P_i$, $P_{i,j} \subset P_j$ and $P_{i,j}$ has a master trigger. For ex: In Figure 20(a), the computation of $x,y,w$ can be considered as 3 independent sub-processes $P_x$, $P_y$, $P_w$ with $\hat{x}$, $\hat{y}$, $\hat{w}$ as master triggers respectively for each sub-process. $P_x$ is the upper part of the process that reads $v$ and outputs the sample of $x$. $P_y$ is the dual of $P_x$ for $u$ and $y$ and $P_w$ is simply the + actor on the right. The clock tree of $P_w$ is a result of intersection of clock trees for $P_x$ and $P_y$. Scheduling such processes requires synchronization constraints and we have to ensure that the schedule does not result in a deadlock. This can be achieved by traversing various branches of clock tree and analyzing the constraints.

If the model satisfies all the above conditions, then the resulting $\{x_1,\ldots x_n\}$ is the set of partial triggers for P. Using these partial triggers and clock trees for the sub-processes, we can generate multi-threaded code by the mapping technique explained in subsection 4.2.3 Concurrent Implementability.

**1) Computing Partial Triggers**: Let $B_p$ represent the system of Boolean equations derived from all the actors present in $P$ . Computing partial triggers for model $P$ is effectively computing prime implicate (non-unitary) for the CNF formula constructed using $B_p$. But computing prime implicate considering entire $B_p$ takes a substantial amount of time. We propose a different approach that computes partial triggers almost two orders faster and is shown in Algorithm 1.

**Algorithm 1**: Compute Partial Triggers Set $S = \{x_1, ..x_n\}$

**Input**: $K$, Set of all signals in model $P$
**Output**: Set of partial triggers $S$ for model $P$
Let $S_p$ be set of possible partial triggers
$S_p \leftarrow \{\}$; $S \leftarrow \{\}$;
**foreach** *Signal* $y \in K$ **do**
  **if** $\nexists z \in K \mid \sigma(z) \supset \sigma(y)$ **then**
    $S_p \leftarrow S_p \cup \{y\}$;
  **end**
**end**
$SubS_p \leftarrow$ all subsets formed using elements of $S_p$
**foreach** $ele \in SubS_p$ *in incremental order of number of signals in* $ele$ **do**
  Set all signals in $ele$ to be absent
  **if** *all signals in* $K$ *can be deduced to be absent* **then**
    $S \leftarrow ele$;
    **return**;
  **end**
  **else**
    $SubS_p \leftarrow SubS_p - \{ele\}$;
  **end**
**end**

Let K be set of all signals in model P , $S_p$ be the set of possible partial triggers and S be the minimal set of partial triggers for P . A signal y ∈ K, cannot be a possible partial trigger if ∃z ∈ K, such that σ(z) ⊃ σ(y). Now we create another set $SubS_p$, which contains all the subsets formed using the elements in $S_p$. So each element of $SubS_p$, is a set of signals. We then select each element ele 2 $SubS_p$, in the increasing order of the number of signals it contains. We set all signals in ele to be absent and check if this implies that the rest of the signals in K are also absent. To set a signal to be absent, we can set the Boolean equation of the corresponding signal to be 0=false. To check for absence, we can see if the other Boolean equations can be deduced to be 0=false. If yes, then S = ele. If no, repeat the procedure with another element of $SubS_p$. At the end of Algorithm 1, S contains the set of partial triggers.

The complexity of Algorithm 1 depends on the complexity of the second for loop, which is $O(2^n)$, where n is the cardinality of $S_p$. We use various techniques to keep the cardinality of $S_p$ to be as small as possible and hence Algorithm 1 completes very quickly even though its complexity is $O(2^n)$. This argument is further strengthened by the experimental results in Table 5 and Figure 22.

**2) Constructing the forest of clock trees T** : In case of sequential code generation, the clock tree has a single root node which corresponds to the master trigger. The child nodes of this clock tree correspond to the signals whose epochs are subsets of the epochs of the signal above them. For the purpose of understanding, this structure can be thought of as a pyramid, where the top of the pyramid corresponds to master trigger of the process and each level below it corresponds to the signals whose epochs can be directly computed if the epoch of master trigger is known. This

levelization is done by repeatedly computing prime implicate of the reduced Boolean formula. This reduced Boolean formula is obtained by setting the boolean variables corresponding to the signals above the current level to true. For example, the signal/s at $n^{th}$ level are obtained by computing prime implicate of the reduced Boolean formula in which all the boolean variables corresponding to the signals in first *n-1* levels are true.

In case of multi-threaded code generation, the clock tree has multiple root nodes which correspond to the partial triggers. The child nodes of this clock tree are derived by recursive prime implicate generation considering one partial trigger at a time. Figure 21(a) shows a pyramid representation of the clock tree in case of a single master trigger and Figure 21(b) shows the same for multiple partial triggers. Algorithms 2 and 3 build the clock tree. The function setTrue(x) produces a reduced Boolean formula that is further used for prime implicate computation. The function setFalse(x) does the opposite, it sets the variable passed in parameter to false. We use it to indirectly select the Boolean formula corresponding to a sub-process: first we set one partial trigger to false ($B_{x=0}$), it marks absence of the partial trigger and all its sub-process, then we complement it ($B_x$=B - $B_{x=0}$) and get back all sub signals of the partial trigger. The function $PI\_Gen_{SMT}$(), takes a Boolean formula in CNF form and outputs prime implicate. For a smaller Boolean formula, the function $PI\_Gen_{SMT}$ () is quite fast and hence we use it in building parts of clock tree. This function uses [1] SMT solver to generate prime implicates as described in [6].

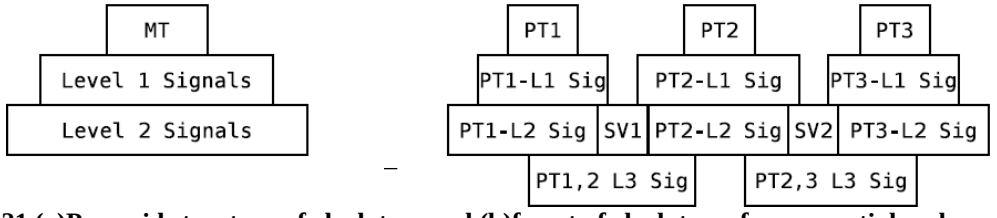



**Figure 21 (a)Pyramid structure of clock tree and (b)forest of clock trees for sequential and concurrent specifications**

**Algorithm 2**: Compute Clock Tree for model $P$

**Input**: Set of partial triggers $S = \{x_1, ..x_n\}$ for model $P$,
System of Boolean Equations $B_p$ for model $P$
**Output**: Entire Clock Tree $T$ for model $P$
Let $N_s$ be a set representing nodes of $T$
Let $E_p$ be a set of all epochs of $P$
$N_s \leftarrow null$;
**foreach** *Epoch* $x_i \in E_p$ **do**
  $N_s$.addNode($x_i$);
**end**
**foreach** *Partial Trigger* $x_i \in E_p$ **do**
  Let $B_{x_i=0} \leftarrow B_p$.setFalse($x_i$);
  Let $B_{x_i} \leftarrow B_p - B_{x_i=0}$; ▷ *Note:* $B_{x_i}$ *is set of Boolean equations for the subprocess of* $P$ *for which* $x_i$ *is the root*
  $N_s$.recNodeOrder($label_{x_i}, x_i, B_{x_i}$);
**end**
**return**

**3) Check for Data Dependencies and Deadlock**: After constructing the clock tree T for model P, we check for cyclic data dependency issues in T . We also check if there are any deadlocks in P. This is done by traversing each branch of the clock tree and analyzing the constraints. If all checks are completed, we conclude that P is concurrent implementable and proceed for identification of shared epochs.

### B. Identification of Shared Epochs

Often signals with different epochs will be involved in some operation (For Ex: Line 5 of Listing 5 where x and y have different epochs). In such cases, epochs of involved signals will be subset of epochs of multiple partial triggers (Ex: In Listing 5, epoch of signal w is subset of epoch of signals x and y). Such signals are said to have shared epochs. Identification of such epochs is important because they correspond to shared variables in software. To compute such shared variables, we need to use synchronization barriers. To identify the signals with shared epochs we use a labeling scheme. Algorithm 3 labels each node in the clock tree with a label that corresponds to the root node under which it is present. All the nodes corresponding to signals that have shared epoch will have multiple labels because they will be under multiple root nodes (For Ex: In Fig 20, node corresponding to epoch of w will have 2 labels - u and v). Rest of the nodes will have single labels.

**Algorithm 3**: recNodeOrder($L, x, B_x$)

**Input**: $L$: Label of a partial trigger node,
$x$: an epoch of Process P,
$B_x$: set of Boolean equations of for a subprocess of $P$ for which $x$ is the root
**Output**: Recursively builds the sub-tree under node $x$
Let $x_N \in N_s$ be the node corresponding to epoch $x$
**if** *isLabelEmpty*($x_N$) **then**
    Let $B_{x=1} \leftarrow B_x.setTrue(x)$;
    $\{PI\} \leftarrow PI\_Gen_{SMT}(B_{x=1})$;
    **foreach** *Epoch* $x_i \in PI$ **do**
        Let $B_{x_i=0} \leftarrow B_{x=1}.setFalse(x_i)$;
        $B_{x_i} \leftarrow B_{x=1} - B_{x_i=0}$;
        ▷ *Note:* $B_{x_i}$ *is set of Boolean equations for the subprocess of* $P$ *for which* $x_i$ *is the root*
        $x_N$.addChildNode(recNodeOrder($label_{x_i}, x_i, B_{x_i}$));
    **end**
**end**
$x_N$.addLabel($L$);
**return** $x_N$

### C. Mapping and Multi-threaded Code Generation

**Algorithm 4**: Code Generation

**Input**: Globals: Model $P$, Clock tree $T$, Set of Partial triggers $S$, Dependency Graph $G$, Shared epochs $E$

**Output**: Multi-threaded Code $MT$

**foreach** *epoch* $x_i \in S$ **do**
  Create newthread $th_{x_i}$;
  $th_{x_i} \leftarrow th_{x_i}$•recCodeGen($x_i$);
**end**
**foreach** *node* $x_i \in E$ **do**
  Create newthread $ths_{x_i}$;
  $ths_{x_i} \leftarrow ths_{x_i} \bullet waitConstraint(labels_{x_i})$;
  $ths_{x_i} \leftarrow ths_{x_i}$•recCodeGen($x_i$);
  $ths_{x_i} \leftarrow ths_{x_i} \bullet notifyConstraint(labels_{x_i})$;
**end**
$MT \leftarrow$main+codeOf($th_{x_0}$+..+ $th_{x_n}$+ $ths_{x_0}$+ .. + $ths_{x_m}$);
**return**

**Algorithm 5**: recCodeGen($x$)

**Input**: Globals: As Algorithm 4, Current node $x$

**Output**: Recursively generate code for all child nodes

Let $C$ represent code for $x$ and its child nodes
$C \leftarrow exportCode(x)$;
**foreach** *child ch of x* **do**
  **if** $numLabels(ch) > 1$ **then**
    $C \leftarrow C$•exportWaitNotifyConstraints($labels_{ch}$);
    **return** $C$
  **end**
  $C \leftarrow C$• Export code for $ch$ and its dependencies
  $C \leftarrow C$•recCodeGen($ch$);
**end**
**return** $C$

After establishing concurrent implementability and build-ing clock tree T , we need to create a mapping that can be used for code generation. Algorithms 4 & 5 give an overview of the code generation procedure. T has multiple root nodes with each root node corresponding to a partial trigger. Each of the partial trigger acts as a master trigger for the corresponding sub-processes, which can be handled by a single thread. So we create and associate a thread ($th_{xi}$ ) for each partial trigger. Now we traverse T in a depth first manner. For each node we visit, we check the number of labels (numLabels()) and the labels it has. The label indicates under which root node/s the current node is present. If it has a single label, then it indicates that its under one root node (not a shared computation). We export the code for this node and append it to the thread

corresponding to the thread pointed by the label (root). Since there are no cyclic data dependencies, we only have to ensure that the input signals to this node are computed before the start of code for the current node. If the node has multiple labels, then it indicates that its a shared computation and we need to wait till the dependencies are computed by other thread/s. We export the wait notify constraints (exportWaitNotifyConstraints()) in the current thread's code and then we handle the shared computation in a different thread. To generate code for the thread handling the shared computation, we start with a wait constraint (waitConstraint()) for the synchronization condition, then we proceed traversing the sub-tree in depth first manner, export code as earlier and finally add the notify constraint (notifyConstraint()). In this way we generate the code for the complete model.

## 4.2.4. Experimental Evaluation and Discussions

We evaluated our proposed approach on the benchmarks listed in Table 4. These benchmarks exhibit either data parallelism or task parallelism or sometimes both. In our evaluation approach, we first manually implemented an efficient C/C++ multi-threaded version of the benchmark using low-level threads. We then modeled the same benchmark in MRICDF and used the tool EMCODESYN (proposed approach) to generate multi-threaded C++ code.

**Table 4: Benchmark Suite**

| No. | Benchmark | Summary of the benchmark |
|---|---|---|
| 1 | Array Addition | Simple data parallel addition. Input is integer arrays of length 10K. |
| 2 | Box Filter | Image processing filter which works by computing the average of surrounding pixels. It exhibits both data and task parallelism. Size of test input is 256x256 pixels (can be any size). |
| 3 | Energy Meter | A model of the control system used in any common home energy measurement instrument. It exhibits task parallelism. In our test suite, we run the system for 3 iterations. |
| 4 | Sieve of Eratosthenes | A prime number sieve for finding all prime numbers up to any given limit (10 million in our example). It exhibits both task and data parallelism. |
| 5 | Tennessee Eastman (TE) Plant-wide Industrial Process [57] | TE process is a simplified model of a real-life industrial process consisting of a reactor – separator – recycler arrangement. In our test suite, we run the TE system for 1 iteration. |

We ensured that the outputs of both versions matched. Finally, we measured the performance of both implementations on a workstation that has 4 Intel Xeon E5405 CPUs with 4GB of memory running Ubuntu 10.10. Performance comparison results are listed in Table 5. Column 4 of Table 5 shows the percentage performance difference between the generated multi-threaded code and hand written multi-threaded code. A negative percentage value indicates that the performance of the generated code is lower than the performance of the hand written multi-threaded code by the corresponding percentage. Experimental results show that the performance of the generated multi-threaded code is almost comparable to the hand-written multi-threaded code. On an average, the generated code for the benchmarks considered is 18.5% slower than the

hand written code. On further analysis, we noticed that this performance difference arises due to,

- Generated code uses a lot of templates as the code generator is implemented keeping a generic application in mind.
- Generated code sometimes creates more threads than actually required. The work done by the separate threads could have been merged and done by a single thread. This additional thread creation and destruction overhead also slows down the performance.

**Table 5: Experimental Results**

| Model Name | Manual Multi-Threaded Performance | | Generated Multi-Threaded Performance | | % Performance Diff. Generated vs Manual | Total Code Generation Time |
|---|---|---|---|---|---|---|
| | LOC | $T_{multi}$ (ms) | LOC | $T_{gen}$ (ms) | (- means Gen. code slower) | (ms) |
| Array Addition | 48 | 12 | 195 | 14 | -16.6 | 428 |
| Box Filter | 96 | 67 | 212 | 74 | -10.4 | 1274 |
| Energy Meter | 215 | 17 | 575 | 18 | -5.8 | 437 |
| Sieve | 56 | 4722 | 178 | 6103 | -22.62 | 1022 |
| TE Process | 613 | 3.5 | 5947 | 4.8 | -37.1 | 2350 |

*LOC stands for Lines of Code, $T_{multi}$, $T_{gen}$ denotes the execution time of the hand-written multi-threaded and generated multi-threaded code respectively.

Theoretically, the scalability of the tool and the proposed approach can be accurately determined when it is applied on a realistic model of a large embedded system (ex: A satellite system). But, modeling such a large embedded system without knowing all the details of the system is not easy. One can also create a large model by duplicating a smaller model. So, we created larger benchmarks by duplicating (2, 4, 8, 16, 32 times) an existing benchmark. The number of inputs, outputs and actors also got multiplied creating the effect of a large embedded system for all practical purposes. Figure 22 shows the time taken for analysis and code generation for these increasing large models. As the models get bigger, there is a linear increase in the time taken for analysis and code generation.

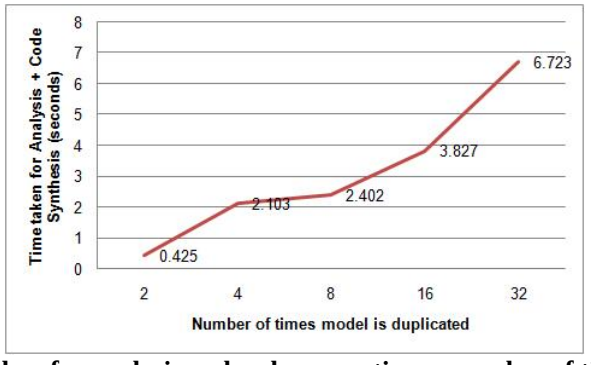



**Figure 22 Plot of Time taken for analysis and code generation vs number of times model is duplicated**

## 4.2.5. Related Work

Numerous efforts have been made in the past to synthesize code from synchronous specifications. But most of these efforts were targeted towards generating sequential code rather than multi-threaded code. Here, we list some of the multi-threaded code generation efforts. The authors of [67] proposed an approach to generate multi-threaded code from Esterel specifications. Their approach involved partitioning of concurrently executable Esterel statements into communi-cating FSMs and distributing the computation of these FSMs based on the communication and synchronization techniques used in reactive processors. In [56], the authors provide a way to translate synchronous guarded actions to multi-threaded C code. They build an action dependency graph using the synchronous guarded actions, extract concurrently runnable tasks from the graph and map them to threads. Both these works are targeted at single clock systems while our work focuses on systems with multiple clocks (polychronous). In [61], the authors provide a non-invasive methodology which includes generating programming glue to generate multi-threaded code from polychronous specifications. This approach requires that, no variables are shared between the concurrently executable processes, in other words, the clock trees of sub-processes do not intersect. This a big limitation and generating multi-threaded code for inde-pendent processes is very trivial. In another similar work [65], the authors focus on generating multi-threaded code for mutually independent tasks, which is trivial. In [66], the authors have explained the concept of weak-hierarchy and composition of endochronous processes. Using these concepts one can identify parts which can be concurrently executed without disabling one another. This work also lists some of the rules for composing endochronous systems to a weakly endochronous system. To the best of our knowledge, there is no implementation of this. The work presented in this article considers and extends the theory presented in [66]. We propose a novel efficient technique by which we can test concurrent implementability of a given MRICDF model by decomposing it. Our technique also generates execution schedule and the multi-threaded code that conforms to the schedule. We present all the algorithms involved and investigate the feasibility and scalability of the proposed technique.

## 4.2.6. Conclusion and Future Work

Writing concurrent programs, especially for safety critical embedded systems, has always been a error prone task. One of the main reasons for this is – immaturity of concurrent programming models as compared to sequential programming models. In this work, we presented a correct-by-construction approach for multi-threaded code gener-ation from formal MRICDF specifications. We presented sound techniques to analyze concurrent implementability of MRICDF models and to generate accurate multi-threaded code. Experiments were conducted to compare the perfor-mance of the generated multi-threaded code against hand written multi-threaded code. We also conducted experiments to test the scalability of the proposed approach and presented the results. In the current version of the tool, the clock tree construction and the code generator implementation are done targeting accuracy and not efficiency of the generated code.

To improve efficiency of the generated code, in future, we plan to apply optimization transformations on the clock tree which can help in generating load balanced code. Mapping of partial triggers to threads might not be the most efficient, especially if the amount of work done

by the thread is not substantially large than thread creation and destruction overhead. In future, we plan to create a thread pool and map partial triggers to tasks. We also plan in future to include formal proofs for all the algorithms and the overall technique.

## 4.3 Synthesizing Embedded Software with Safety Wrappers through Polyhedral Analysis in a Polychronous Framework

We investigated the use of various decision making tools to check these properties. First we looked into SAT Modulo Theory (SMT) solvers and later on we looked into Polyhedra libraries. Below we explain each work with their advantages and limitations.

### 4.3.1 SMT based safety property checking

In this work we show how one can use SMT solvers for checking of a particular safety property – causal loop detection. The approach is generic and can be used to verify most of the properties. In earlier works [10], causal loop detection was done by generating SMT equations for the entire MRICDF model and this set of equations was given as an instance for the SMT solver. The disadvantage is that for a large scale example, the SMT instance will become huge and can lead to long run times – sometimes never ending. In our work, we first mine the specifications for possible causal loops. We then express the clock constraints of the dependencies as SMT equations and check if all the equations can be true at same time or not by evaluating the SMT instance. If the SMT instance evaluates to *true*, then there exists a causal loop, otherwise no.

We now illustrate this work with an example. For the reason of expressiveness, we are using Signal and *Polychrony* here instead of MRICDF and *EmCodeSyn*. Consider the Signal code shown in Listing 6. We compile this code using *Polychrony* to check for presence of possible causal loops. From the code in Listing 6, we can observe that, when *isMin* is true, then *avg* depends on *max* and *max* depends on *avg* causing a true causal loop. Similarly when *isMin* is false, then *avg* depends on *min* and *min* depends on *avg* causing another true causal loop. Once we identify possible causal loops, we mine the information regarding clock constraints leading to the possible causal loops. After mining we encode the clock relations as SMT equations and construct a SMT instance and test it for satisfiability. Clock relations are shown in Listing 7. It can be noticed that there are two sets of clock relations showing two possible true causal loops.

**Listing 6: Constructive Causal Loop**

```
process causal -smt =
(? integer initial , step 1, intMin ;
! integer min , avg , max;
)
(| initial ^= step 1 ^= intMin ;
|min := initial when intMin <5 default avg - step 1
|avg := min+ step 1 when intMin =10 default max - step 1
|max := avg + step 1 when intMin >10 default initial
|);
```

**Listing 7: Clock relations**

```
% Loop 1
(|{ avg --> max } when C_CLK _0
|{ max --> avg} when C_ CLK _1
|)
% Loop 2
(|{ min --> avg } when C_CLK _2
|{ avg --> min} when C_ CLK _3
|)
where , C CLK 0 := :(intMin = 10), C CLK 1 := intMin > 10,
```

```
C CLK 2 := :(intMin < 5) and C CLK 3 := (intMin = 10)
```

Any constraint solver enriched with integer theories can be used. In our work, we have used the latest YICES SMT solver [44], as a constraint solver. Translating the above clock relations as YICES input, we get the equations in Listing 8. Invoking YICES on these equations will give a SAT result as explained earlier. Also YICES provides a counter example (*intMin*=11 & *intMin*=10) where the constraint is satisfied which matches with our earlier interpretation. Hence there exists true causal loops in the specification shown in Listing 6 and one possible way they can be formed is when *intMin*=11 & *intMin*=10. If YICES had given an UNSAT result, then we conclude that the property is not satisfied and hence it is a false causal loop. Similarly any safety property can be expressed as SMT instance and verified.

**Listing 8: SMT equations for Loop 1**

```
;; Loop 1
( define intMin :: int)
( assert (and (not (= intMin 10)) (> intMin 10) ) )
( check )

Result :- sat (= intMin 11)

;; Loop 2
( define intMin :: int)
( assert (and (not (< intMin 5)) (= intMin 10) ) )
( check )

Result :- sat (= intMin 10)
```

**Limitations of this approach:**

Safety property verification such as causal loop detection is not a trivial problem. Given a data dependency loop, the complexity of checking if it's truly causal or not is at least NP-hard. If all inputs are Boolean signals, and dependencies can be expressed as Boolean functions using ANDs and ORs and NOTs, then the problem would be the same as solving a SAT instance and is NP-Complete. But if the dependencies can be expressed as arbitrary functions over integers or reals or other complex data types, then the problem is undecidable. This shows that any method must be based on heuristics and are likely not complete. One must strive for as close to complete a solution but never compromise on *soundness*. This is what we have tried to achieve in this work. Also in this work, we only handle non-floating point and linear constraints. This is because of the limitation of YICES and not of the approach. Another disadvantage of this work is that if a property fails, the tool will just output one of the many possible scenarios when the property will fail and not all the scenarios when property fails.

## 4.3.2 Polyhedra based safety property checking

In this work we try to preserve the advantages of SMT based approach and try to address its disadvantages. If the synthesized software has to interact with a physical environment, often additional range constraints on various inputs as well as outputs are provided. Analyzing the safety of execution often leads to analysis of reachability, invariants, and cyclic dependencies which may be affected by such range constraints. As explained in the last paragraph, while analyzing a specification for a safety property, even if it violates an invariant property, or shows cyclic dependency – in a very limited area of its reachable state space, it will be rejected totally.

For such specifications, instead of rejecting the specification outright, the synthesis tool should guide the user by showing the exact ranges of the input values (or equational relationships between the inputs as appropriate) that could direct the resulting program to such violating area of the state space. This is exactly the problem we address in this work. To make decisions with range constraints, we use Polyhedral libraries as they can take affine relations as constraints.
We now illustrate the problem being addressed in this work with an example.

**Listing 9: Causal Loop Example**

```
process AC_ DISPLAY = (? integer minT , curT , maxT ;
! integer disp _coldT , disp _hotT , disp _ normT )
(| minT ^= curT ^= maxT
| disp _ coldT := minT when curT <70 default curT
| disp _ normT := ( disp _ coldT +5) when curT =70 default
( disp _hotT -5)
| disp _ hotT := ( disp _ normT +5) when curT >80 default maxT
|);
```

Consider the example shown in Listing 9. A Boolean abstraction based check would replace each predicate by a Boolean variable taking arbitrary values, and will not consider the relationship between the predicates in their numerical domain. As a result a causal dependency loop will be detected by such analysis because of the interdependency between *disp_normT* and *disp_hotT*. However, if our abstraction is cognizant of a theory of integers with ordering relations, then it would lower the Boolean abstraction to a model that considers intervals with ordering. On this model, one could prove that when $curT>80$, only then such causal dependency loop will exist. Obviously, if this happens, the system will behave non-deterministically or will deadlock. If this information is explicitly presented to the user upon completion of the analysis, and the user can guarantee an additional input constraint, $70\leq curT\leq 80$, then generating code from this specification is completely legitimate – as the program will not display any deadlock behavior. In addition, if one wants to ensure safety, one could produce a wrapper that would intercept all inputs *curT* and check against this constraint, and filter out any occurrence of input value that violates the user guaranteed constraints. However, if the user can guarantee only $70\leq curT\leq 90$ – the system will exhibit causal behavior when $80<curT\leq 90$. But the system has a safe operating area, $70\leq curT\leq 80$. One could still apply a wrapper to prevent the system from moving outside its safe operating area – if it makes sense for the application.
We propose a polyhedral model based causality analysis technique which can accept Boolean, integer and rational input constraints and check for violation of safety properties (e.g., existence of causal loops) in the constrained system. Based on polyhedral analysis of the constraints and specifications, we also propose a technique to identify the safe operating area of the system in terms of bounds on input and other linear constraints. In case of multiple safe operating areas, our technique lists all of them. We also propose a safe code synthesis technique by adding wrappers to ensure that the resulting system does not behave non-deterministically or deadlock even when the input constraints are accidentally violated.

We illustrate our solution with an example now. Consider the signal program shown in Listing 10, which is an extension of the program shown in Listing 9.

**Listing 10: True Causal Loop**

```
process AC_ DISPLAY = (? integer minT , curT , maxT , curP , curK
! integer disp _coldT , disp _hotT , disp _ normalT )
(| minT ^= curT ^= maxT ^= curP ^= curK
% Conditions %
| cond _1 := (( curT >= 2) and ( curT <= 18))
| cond _2 := (( curP >= 3) and ( curP <= 21))
| cond _3 := (( curK >= 25) and ( curK <= 35))
| cond _4 := (curT - curP >= -10)
| cond _5 := (( curT + curP >= 11) and ( curT + curP <= 33))
% Output Computation %
| disp _ coldT := minT when (curT < minT ) default curT
| disp _ normalT := ( disp _ coldT +10) when
(not( cond _1 and cond _2 and cond _3))
default ( disp _hotT -10)
| disp _ hotT := ( disp _ normalT +10) when ( cond _4 and cond _5)
default maxT
|)
where
boolean cond _1, cond _2, cond _3, cond _4, cond _5;
end;
```

When a Boolean abstraction is analyzed, it identifies the possibility of causal loop because of the interdependency between *disp_hotT* and *disp_normalT* as shown in Listing 11. One can invoke an SMT solver to check for nullity of clock constraints (*C_CLK*_31∧*C_CLK*_23) on the path of the apparent loop. This is done by extracting the clock constraints and generating the predicates for Yices SMT solver as shown in Listing 12. Invoking Yices solver will decide this condition as *satisfiable* (which indicates the existence of true causal loops) and it outputs *one* counter example to show a case where causal loop may create a deadlock. If we include input constraints, an SMT solver will not be able to provide us the safe operating region of the input space.

**Listing 11: Possible Causal Loop**

```
(| { disp _ hotT --> disp _ normalT } when C_ CLK _31
| { disp _ normalT --> disp _ hotT } when C_CLK _23
|)
where , C_ CLK _31 = cond _4 and cond _5
C_CLK _23 = cond _1 and cond _2 and cond _3
```

**Listing 12: Assertion in SMT solver and Solution**

```
( define curT :: int) ( define curP :: int) ( define curK :: int)
( assert (and (<= curT 18) (<= curP 21) (<= curK 35)
(>= curT 2) (>= curP 3) (>= curK 25) (<= (+ curT curP ) 33)
(>= (- curT curP ) -10) (>= (+ curT curP ) 11) ) )
( check )

Result : SAT , Counter example : curT =8, curP =3, curK =25 %
```

## Constraint Extraction and Transformation for Polyhedral analysis

Let us say we are given the input constraints shown in column 1 of Table 6 for the SIGNAL program shown in Listing 9. The clock constraints for possible causal loop are also transformed to a system of affine inequalities and equations. They are shown in column 2 of Table 6. There exists an implicit logical intersection among all the constraints within each column of Table 6. The constraints in Table 6, needs to be transformed into affine form to use the *PolyLib* library [51]. The system of translated affine inequalities are shown in Table 7. This system is further abstracted to matrices before using Polylib APIs.

**Table 6 Input and True Causal Loop constraints**

| Input Constraints | Loop Constraints |
|---|---|
| $10 \leq \text{curT} \leq 40$ | $2 \leq \text{curT} \leq 18$ |
| $10 \leq \text{curP} \leq 40$ | $3 \leq \text{curP} \leq 21$ |
| $10 \leq \text{curK} \leq 40$ | $25 \leq \text{curK} \leq 35$ |
| | $\text{curT} - \text{curP} \geq -10$ |
| | $11 \leq \text{curT} + \text{curP} \leq 33$ |

Figure 23 shows the plot of polyhedras representing both input constraint and true causal loop constraints. From multiple views we see that there exists a region of intersection between the two polyhedras, which indicates the existence of true causal loops with the current input constraints.

**Table 7 Inequalities and Equations from Input and Loop constraints**

| Input | Loop |
|---|---|
| $\text{curT} - 10 \geq 0$ | $\text{curT} - 2 \geq 0$ |
| $-\text{curT} + 40 \geq 0$ | $-\text{curT} + 18 \geq 0$ |
| $\text{curP} - 10 \geq 0$ | $\text{curP} - 3 \geq 0$ |
| $-\text{curP} + 40 \geq 0$ | $-\text{curP} + 21 \geq 0$ |
| $\text{curK} - 10 \geq 0$ | $\text{curT} - \text{curP} + 10 \geq 0$ |
| $-\text{curK} + 40 \geq 0$ | $\text{curT} + \text{curP} - 11 \geq 0$ |
| | $-\text{curT} - \text{curP} + 33 \geq 0$ |
| | $\text{curK} - 25 \geq 0$ |
| | $\text{curK} + 35 \geq 0$ |

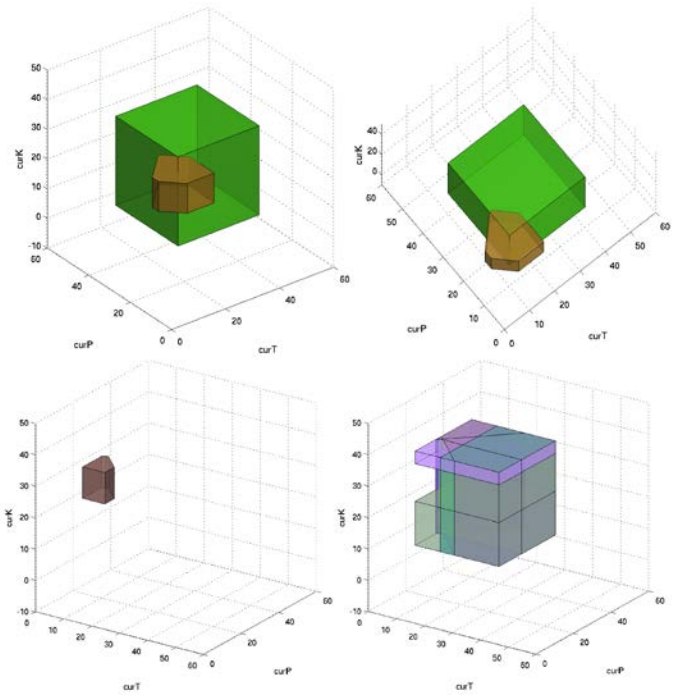



**Figure 23: (Top) 3D-plot (multiple views) of Polyhedras representing Input and Loop Constraints. (Bottom) 3D plots of I ∩ L and I - L**

**Polyhedral Analysis**

To obtain the bounds of safe operating region and the region where true causal loop exists, we apply two polyhedral operations from the *PolyLib* library.

i *DomainIntersection*(*I*,*L*): This operation returns the intersection of two polyhedral domains. This is used to compute $I \cap L$.

ii *DomainDifference*(*I*,*L*): This operation returns a new polyhedral domain which is the difference, $I - L$.

Both these operations may return many sub-polyhedras instead of one single resultant polyhedra. Union of all the sub-polyhedras will give the resultant polyhedra. Figure 23 also shows the plots for both $I \cap L$ and $I - L$ respectively. One has to observe that the plot of $I - L$ actually is a union of 6 different polyhedras.

**Limitation of Polyhedral libraries**

Almost all of the existing polyhedral libraries including the one we are using, *PolyLib*, have restrictions that they can only accept integer constraints. In our technique, all rational constraints are multiplied by least common multiple to obtain integers, and floating point numbers are truncated based on precision specified by the user. Then we multiply the truncated floating point constraint by a suitable number such that it becomes an integer.

**Safe code synthesis using Wrapper**

From the result of polyhedral analysis, we obtain the bounds on inputs for safe operating region that we must check before actually passing it to the process so that the process remains in safe trajectories. Then a wrapper code is inserted which prevents any inputs violating the conditions of safety from being passed. The user of the synthesis tool is given an option to choose

if such implementation makes sense in the application domain. In Listing 13 we show the wrapped code for the SIGNAL program shown in Listing 9.

**Listing 13: Signal program of Listing 12 with wrappers**

```
process AC_ DISPLAY = (? integer minT , curT , maxT ;
! integer disp _coldT , disp _hotT , disp _ normT )
(| minT ^= curT ^= maxT ^= cond _1
| cond _1 := (( curT >= 70) and ( curT <= 80))
| disp _ coldT := ( minT when curT <70 default curT ) when cond _1
default DEFAULT _ VALUE
| disp _ normT := ( ( disp _ coldT +5) when curT =70 default
( disp _hotT -5) ) when cond _1
default DEFAULT _ VALUE
| disp _ hotT := (( disp _ normT +5) when curT >80 default maxT )
when cond _1 default DEFAULT _ VALUE
|)
where
bool cond _1;
end;
```

## 4.4 Real-Time Extension and Improved Schedulability Analysis for Real-time Code Generation from Polychronous Specifications

Verifying the hand-written real-time software for complex safety critical applications is difficult and time consuming. Testing and simulation techniques are not easily scalable and are non-exhaustive. Formal code generation tools have been effectively employed for such purposes. These tools accept formal specifications of a complex system and not only generate bug-free code, but also guarantee certain safety properties in the generated code. Prelude, is one such formal language which can be used to specify complex real-time systems. During schedulability analysis, Prelude compiler - *preludec*, over-approximates and considers certain conditional tasks to execute always. This technique though sound is imprecise, which may at times lead to over-approximation of worst-case execution time (WCET) for the system and rejection of the certain set of tasks as un-schedulable, though they can be scheduled in reality. In this work, we first propose real-time extensions to MRICDF, a formal polychronous programming language. We then adapt the extended precedence encoding technique of Prelude and improve its existing schedulability analysis techniques for multi-periodic real-time systems by considering the occurrence conditions of tasks. This improved schedulability analysis provides a tighter WCET and expands the domain of schedulable tasks.

### 4.4.1 Introduction

As real-time embedded control systems applications become larger and more complicated, and the platforms on top of which they run become more complex and diverse, it becomes increasingly difficult for software developers to manage all details of a design. Most embedded control systems are first developed at high level of abstraction using control theory and other techniques and often times the implementation details are left to the software developer to manage. Development of software at this level can be very error prone, can make porting the design to different platforms difficult, and may not guarantee determinism or any safety properties.

There have been many attempts in the past to address these aforementioned issues within works such as Lustre with real-time extensions[94], Simulink with Real-Time Workshop [103], and Prelude[93]. However with some of these works, attempts to formalize the language and model real-time systems have been an after thought. Prelude is a synchronous language that specifically targets real-time systems with a formally defined semantics. The language and corresponding compiler *preludec* provide means to specify multi-periodic real-time systems, perform variety of static analysis techniques and compile the specifications into real-time code for their real-time operating system, SchedMCore. One of the important static analysis technique performed on Prelude specifications is Schedulability Analysis. This determines whether a model will meet all deadlines over its execution. One short coming of *preludec* is that during schedulability analysis, it ignores the conditions on which tasks has to execute and assumes that the tasks will always execute. This over approximation of execution of conditional tasks tasks within their system [97] and leads to the possibility of looser WCET bounds and certain tasks being incorrectly rejected as unschedulable.

In this work we will explain further the Prelude language, and how the schedulability analysis is performed as well as show two potential solutions to this over approximation technique. Finally we will show how these two solutions can be used to develop real-time systems using the synchronous language, MRICDF, and development framework, EmCodeSyn.

#### 4.4.1.1 Motivation

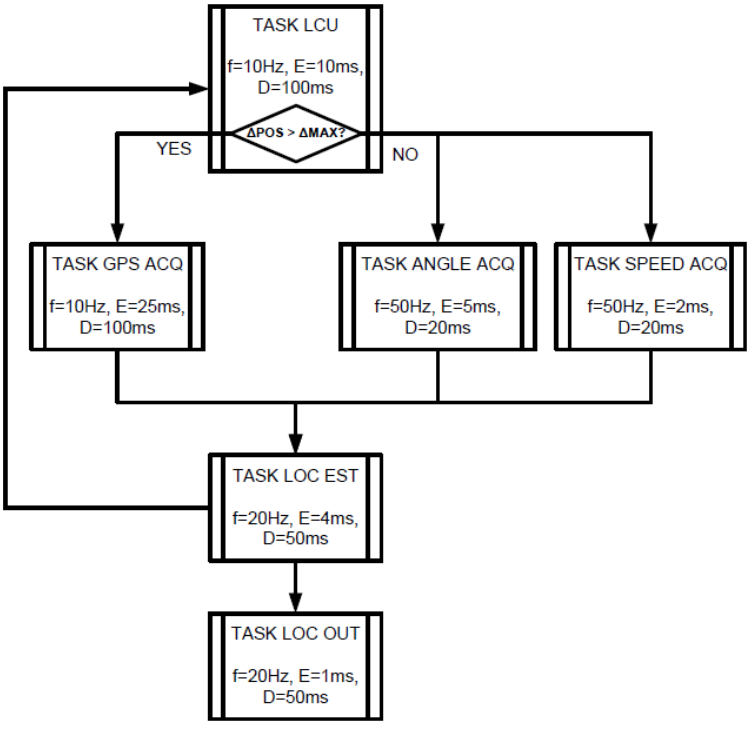



**Figure 24: Location Estimation Unit (f=Rate of occurrence, E=Execution Time, D=Deadline)**

Consider the data flow model of a system shown in Figure 24. This system represents a location estimation unit that has two main data acquisition modes: obtaining many velocity vectors from direction and speed sensors (Task ANGLE ACQ and Task SPEED ACQ) to estimate position, or obtaining GPS coordinates (Task GPS ACQ) for a more precise location. The location (Loc.) EST task then uses whichever data was obtained to calculate a location and sends that value to the Loc.OUT task which outputs the value. Loc. EST also determines the error inherent in the measured value and returns that value to LCU task. In order to determine which acquisition mode should be used, the LCU uses the error value stored in a buffer from Loc. EST to determine if the current uncertainty value $\Delta Pos$ has crossed a threshold given as $\Delta max$ and GPS ACQ must be triggered to return the uncertainty of the estimated location to a safe level. Also shown in the figure are the frequency, execution time and deadline of each task.

A real time scheduling problem usually consists of asking the scheduler whether a feasible schedule exists for a given set of tasks and a stipulated amount of computing time. In our example, we ask the *preludec* tool, if a feasible schedule exists for the given set of tasks in 100 ms of computation time. Looking at this system, it is obvious if the conditions on the communications from task LCU are considered, there are 2 possible schedules –

(a) 1 instant of LCU, 2 instants of Loc. EST and Loc. OUT, and 5 instants of SPEED ACQ and ANGLE ACQ,

(b) 1 instant of LCU, 2 instants of Loc. EST and Loc. OUT, and 1 instant of GPS ACQ.

The *preludec* tool ignores the conditional communications originating from LCU because of its inability to statically determine task activations and instead are over approximated to be always active. Thus, *preludec* tries to schedule all the tasks within 100 ms of computation time and then states that its un-schedulable. But theoretically, there exists 2 possible schedules.

While the activations of such task communication clocks may not be statically determined, there are inferences that can be drawn from the model that can create a more refined view of the system and tasks that can result in fewer rejections of schedulable systems.

#### 4.4.1.2 Contribution

Specification of an system in a polychronous language such as MRICDF, will provide us with the ability to do high level analysis of logical clocks. But without the real-time features, MRIDF cannot be used directly to specify real-time systems. Thus, we first propose real-time extensions to the formal MRICDF language and the development environment EmCodeSyn that will allow for the specification of real-time characteristics of systems, such as tasks, execution times, deadlines, etc.

We then adapt and extend the schedulability analysis found in *preludec* tool. We introduce the concept of conditional task graph and use this graph to explore the execution space of the model. We use the exploration results in improving the schedulability analysis there-by determining a more refined WCET for the system and also expand the domain of schedulable tasks by reducing the false negatives. We compare the results of the proposed technique with the results of *preludec* tool.

## 4.4.2 Intro to Prelude

Prelude is a formal language used in the development of real-time embedded systems. It is in the family of data-flow languages such as Signal [98] and MRICDF [100], but specifically focuses on defining of software architectures for multi-rate, multi-periodic systems. Prelude does automated translation of the multi-rate software into a real-time software implementation and verifies certain safety and temporal properties at compile time.

In this section we will discuss some of the basic clock theory that Prelude uses to generate deterministic real-time software from specifications. We will also cover some of the static analysis that is performed before code generation; speficially, we will discuss precedence encoding and schedulability analysis.

#### 4.4.2.1 Periodic Clocks

The Prelude synchronous real-time model relies on the Tagged-Signal Model [96]. In this model and similar to other synchronous languages, variables and expressions are represnted as *flows*. A pair, $(v_i,t_i)_{i \in N}$, where $v_i$ is a value in the domain $V$ and $t_i$ is a date in $Q$, $\forall t_i \in Q, t_i < t_{i+1}$, can be used to represent the value of a variable or expression at a specific date $t_i$. A flow is then a sequence of these pairs and represents a variable or expression value over the set of all dates. The

*clock* of a flow is then a set of dates in *Q* in which the value $v_i$ must be computed, and the value $v_i$ must be computed $[t_i, t_{i+1}[$, or one *instant*. This means that flows may have different dates for when $v_i$ must be computed, and thus can have different clocks as well as different instant durations. With different clocks comes a variety of relations that can be defined between such clocks and flows such as equivalence; two clocks are equivalent if they active for all of the same dates. Prelude focuses on a specific subset of clocks called *strictly periodic clocks* [97]:

**Definition 1** *(Flow, Flow Clock, Flow Instant) A flow, f is a sequence of pairs,* $(v_i,t_i)_{i \in N}$*, where* $v_i$ *is a value in the domain V and* $t_i$ *is a date in Q,* $\forall t_i \in Q, t_i < t_{i+1}$*. The flow clock, ck(f), is the set of dates in Q where one value* $v_i$ *must be computed. A flow instant is one date in the flow clock.*

**Definition 2** *(Strictly periodic clock). A clock* $h=(t_i), i \in N, t_i \in T$*, is strictly periodic if and only if:* $\exists n \in Q, \forall i \in N, t_{i+1} - t_i = n$, where *n* is the *period* of *h*, denoted $\pi(h)$, and $t_0$ is the *phase of h, denoted* $\varphi(h)$.

Strictly periodic clocks are then able to define a flow's instants in terms of a rational valued real time clock, by giving the period and phase, while the Boolean clock of a flow gives the activation condition of a flow for a specific instant. Strictly Periodic clocks offer a way to compare and transform different clocks that are not offered with Boolean clocks alone. This subset of Boolean clocks can be compared via their period and phase characteristics. Transformations on these clocks can be done as well to alter their characteristics. Three strictly periodic clock transformations are defined [96]:

**Definition 3** *(Periodic clock division). Let* $\alpha$ *be a strictly periodic clock and* $k \in Q$*."*$\alpha/.k$*" is a strictly periodic clock such that:*

$$\pi(\alpha/.k) = k * \pi(\alpha), \varphi(\alpha/.k) = \varphi(\alpha)$$

**Definition 4** *(Periodic clock multiplication). Let* $\alpha$ *be a strictly periodic clock and* $k \in Q$*."*$\alpha *.k$*" is a strictly periodic clock such that:*

$$\pi(\alpha*.k) = \pi(\alpha)/k, \varphi(\alpha*.k) = \varphi(\alpha)$$

**Definition 5** *(Phase offset). Let* $\alpha$ *be a strictly periodic clock and* $k \in Q$*."*$\alpha \rightarrow .k$*"is a strictly periodic clock such that:*

$$\pi(\alpha \rightarrow .k) = \pi(\alpha), \varphi(\alpha \rightarrow .k) = \varphi(\alpha) + k*\pi(\alpha)$$

The Prelude software model consists of real-time tasks and communication between tasks. In order to produce deterministic code from these models it is a requirement to have formally defined communication operators. The periodic clock transformations give three basic operators along with an instant delay operator: multiplication, *^, division /^, delay *fby*, and phase shift:>. These operators are used to equate two flows. For example, a flow *f* with a period that is one third of another flow *g* the expression */^3* would equate the two flows.

Although these operators can equate two flows they must also be deterministic. In the previous example if *f* is writing to *g* then it must be known for any instant of *g* which instant of *f* it is dependent on. A function, $g_{ops}(n)$, describes this relationship. In this function *ops* is the rate transition operator and *n* is the specific instant of the independent flow. In our example $f \xrightarrow{/^3} g$,

$g_{/\wedge 3}(n)$ will describe the deterministic data relation relation between these two flows. The function $g_{ops}(n)$ is inductively defined below [96]:

$$g_{*\wedge k.ops}(n) = g_{ops}(kn)$$
$$g_{/\wedge k.ops}(n) = g_{ops}(\lceil n/k \rceil)$$
$$g_{\sim>q.ops}(n) = g_{ops}(n)$$
$$g_{fby.ops}(n) = g_{ops}(n+1)$$
$$g(n) = n$$

When two clocks are equivalent there is no transition operator necessary; shown as $g(n)=n$. This means that the $n^{th}$ instant of the producer flow $f$ is consumed by the $n^{th}$ instant of the consumer flow $g$, or $f(n) = g(n)$. Other transitions are defined in terms of this basic function, where more complex operators can be defined from the composition of these basic operators. For example, if flow $g$ has a period that is half that of $f$ the rate transition is *^2 and $g*\wedge 2(n)=2n$, or $f(n)=g(2n)$. Every other instant of $g$ consumes an instant of $f$ because of the difference in periods. A few examples of different rate transitions and corresponding $g_{ops}(n)$ functions between flows $f$ and $g$ can be seen in Figure 25.

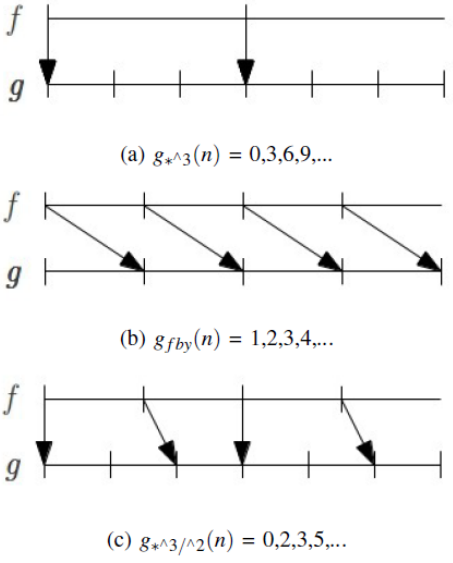



**Figure 25: Variety of ops rate transitions between flows f and g**

Prelude also includes Boolean clock operators; *when* and *whennot*. These operators do not affect the temporal characteristics such as period or phase of a clock. Instead they give certain conditions under which the clock is present or not present. This means that a clock *f whenc* for a flow *f* and Boolean condition *C* will have the same period and phase as *f* but will only be present in an instant if the condition *C* is true.

These operators allow a user to express a variety of deterministic communication structures between flows. When these are combined with tasks a high level task graph abstraction can be formed.

#### 4.4.2.2 Task Graph

In order to perform static schedulability analysis the text present in the user defined Prelude process must be translated into a *task graph*. A task graph is a collection of vertices, where each vertex represents a task, and each edge represents a precedence, or data dependence, relationship. The first step to create a task graph is to expand the original process. This is done recursively by replacing intermediate expressions or vertices with an equivalent set of more basic vertices. This is done until the only expressions left are user defined or imported functions. This collection of base functions or expressions create the tasks within a process[96].

In [96], the authors provide the details of translating the expanded program into an intermediate graph and then how they reduce the intermediate graph into the final task graph. For our purposes it will suffice to understand the structure of the Prelude task graph. A Prelude process can be represented with a graph $g=(V,E)$, where $V$ is a set of vertices or tasks and $E$ is a set of edges or precedences. Each vertex $v_i \in V$ contains a set of characteristics $(in_i, out_i, f_i)$, where $in_i$ is the set of task inputs, $out_i$ is a set of task outputs, and $f_i$ is the relation between the inputs and outputs. Precedences or edges occur when there is a variable $v$ such that $v \in out_i$, $v \in in_j$ and the precedence is represented as $t_i \rightarrow t_j$.

A vertex $v_i$ represents a task $t_i$ which has real-time characteristics. These are represented as $(T_i,C_i,r_i,d_i)$. Tasks are treated as synchronous blocks, all inputs and outputs have the same periodic clock, $pck_i$. This clock can be used to derive the period, $T_i = \pi(pck_i)$, as well as the release date or phase, $r_i=\varphi(pck_i)$, of the task $t_i$. The other two characteristics $C_i$, worst case execution time, WCET, and $d_i$, deadline, are derived from user specifications. There is no analysis done to determine the WCET of a task. Instead the user provides this value by specifying execution times within the process. The deadline, $d_i$, is by default $d_i=T_i$ but if the user has specified a deadline for any output in $out_i$ then $d_i$ is the minimum specified deadline for all outputs in $out_i$ [96].

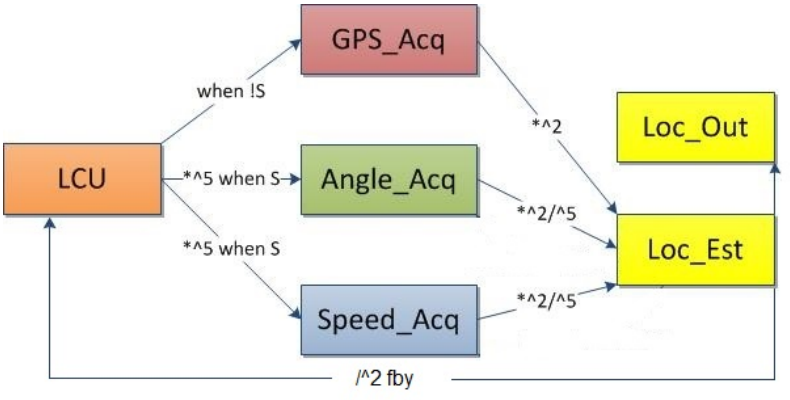



**Figure 26: Prelude Task Graph for Location Estimation**

The task graph for the Location Estimation example discussed previously can be seen in Figure 26. This task graph shows the proper rate transition operators between each task precedence relation. Here we are assuming that $S$ is the condition where $\Delta Pos \leq \Delta max$. It also includes the buffer operator that exists for the position feedback.

#### 4.4.2.3 Static Analysis

From the task graph abstraction, Prelude performs a variety of static analysis to validate the model for both safety and temporal properties. One major tanslation that is done is *extended precedence encoding*. Extended precedence encoding returns a set of indepedent periodic tasks with no precedence relations. This is done by adjusting the characteristics of each task in a way that guarantees the precedence relation will exist within the operation of the real-time system [96]. This is an important translation because most schedulability analysis tools are unable to determine a schedule for a graph with multi-rate precedences [104]. Although this translation allows for a schedule to be generated for the model, it may not accurately reflect the execution of the software during run time.

When determining the execution of a model it is important to understand what conditions are needed for each task to execute. Prelude defines this as the *activation condition* of a task. An activation condition, $cond_i$, is a Boolean formula describing the conditions under which the task $t_i$ will execute. This is important when there are Boolean operators on rate transitions such as *when* and *whennot*. Let *cond()* be a function that determines if a given flow is present for an instant, where *pck* is a periodic clock and *c* is a Boolean condition. The presence or absence of an input can be seen below:

$$cond(pck) = true$$
$$cond(pck\ when\ c) = cond(pck) \wedge c = c$$
$$cond(pck\ whennot\ c) = cond(pck) \wedge !c = !c$$

The activation for the activation condition of task $t_i$ is then the disjunction of *cond()*for every input clock, *ck*, of a task:

$$cond_i = \bigvee_{ck \in ins_i} cond(ck)$$

An important note when discussing the precedence encoding or static analysis techniques of Prelude is that the Boolean operators *when* and *whennot* are overapproximated during static analysis[97]. These operators are still included when generating behaviorally equivalent code from the given model, but the Boolean conditions for these operators are simplified to be always true for static analysis. This also means that *cond(pck whenc)* is always true and that tasks are assumed to always execute during static analysis.

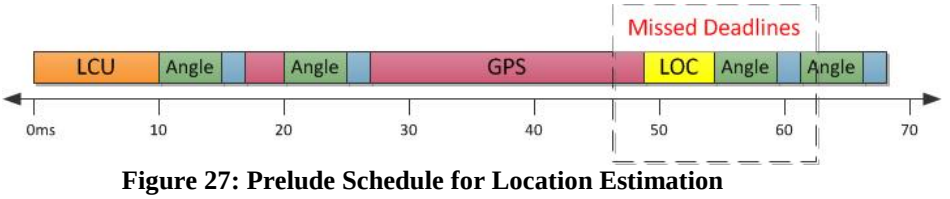



**Figure 27: Prelude Schedule for Location Estimation**

This overapproximation can create false negatives; models being rejected as unschedulable when in fact that meet timing constraints during run-time. The Location Estimation example is one such model. If the Boolean operators are ignored and every task is executed then this model becomes unschedulable. This can be seen in Figure 27.

This model is in fact schedulable though because GPS_Acq and Angle_Acq or Speed_Acq are mutually exclusive tasks determined by the Boolean operators. What we will propose in the next section is a technique to use the Boolean operators to determine a more refined execution schedule. By using a more accurate execution model for static analysis we will avoid some of the false negatives that are present when the Prelude overapproximations are made.

### 4.4.3 Conditional Task Graph

Prelude translates a task graph *g* into a set of indepedent tasks *q*. They do this through extended precedence encoding which alters the task characteristics, such as deadline and release date, to guarantee that the precedence holds during execution. This set *q* is then used for schedulability analysis. We refine this set *q* further via the use of a conditional task graph, or CTG, to reduce the propensity of false negatives during schedule analysis.

The conditional task graph has an equivalent structure to the graph *g*. This means that there are the same tasks with the same precedence relations that exist. We use the updated task characteristics that were obtained through extended precedence encoding and we simplify the rate operators to only include Boolean operators. Instead of using this graph to verify precedence relations during execution, the CTG will be used to explore the possible ways the tasks in a real-time system will execute during run-time.

Before discussing the methods to explore the execution space of a conditional task graph *c*, we will define a few terms to describe such executions.

**Definition 6** *(Task Instants) Each task $t_i$ within c is a synchronous block. This means that the periodic clocks of all flows within a task are equivalent. The task instants, $\Upsilon_i$ , of $t_i$ is the maximal set of instants in which every flow in $t_i$ can be computed and it represented by the clock $c_i$.*

A task instant is a logical event where the activation condition, $cond_i$, must be computed. If the activation condition is true, meaning the clock of any of the inputs signals to the task is true, then the task must be computed. The maximum amount of time that the computation will take in continuous time is given by the WCET of the task. We refer to any logical instant where the activation condition signal is true, to be a *task activation* and a computation of WCET duration must occur at this instant.

**Definition 7** *(Task Activation)Let $\Upsilon_i$ be the set of task instants, which represents the set of logical events in which the task $t_i$ may be computed. The $t^{th}$ instant in $\Upsilon_i$ where $[cond_i(t)]$, is a task activation, denoted $a_i(t)$, where $a_i$ is the activation flow. The set of all activations of $t_i$ is denoted $A_i$. $A_i \subseteq \Upsilon_i$.*

A task activation denotes the computation of a task at a specific logical time. However, within this activation not all flows of the task will be computed. Because these tasks are treated as synchronous blocks, determining which flows are actually computed within a task is not consequential to the execution time of that task. If the task executes then it always executes for the amount of time denoted by the WCET of that task. Although the flows within a task do not

change the execution time, they can be used to determine whether an output is generated from a task for that given activation.

For a task $t_i$, there is a finite set of output combinations that can occur during one task activation. These outputs obviously determine what other tasks within $c$ will receive inputs, so it is required to define these combinations. For determining the worst case execution of a model we only need to conern ourselves with the maximal output combinations. Such combinations we will call *branches*.

**Definition 8** *(Branch) For any vertex or task $t_i$ within $c$, $out_i$ describes the set of task output flows. Each flow has the same strictly periodic clock but can potentially have differing Boolean clocks. For any set of output flows $out_i$ there exist a subset of minimal flows $mins \subseteq out_i$, such that $\forall f \in mins, \nexists g \in out_i, g \subset f$. A branch, $br_f$ is then a set of output flows that must be present in one task activation given the presence of a minimial flow $f \in mins$.*

A branch, $br_f$ can be seen as a grouping of outputs that must occur together and each task $t_i$ has a set of branches, $Br_i$, that describe these maximal groupings of outputs. The logical instants when these outputs are present can be described in terms of a flow which is referred to as a *branch flow*. A branch flow, $bf_f$ is a flow whose clock is equivalent the minimal flow $f$ within a branch, $br_f$.

**Definition 9** *(Branch Flow, Branch Activation) For any branch $br_f$ within a branch set $Br_i$ of task $t_i$, the branch flow $bf_f$ is a flow whose clock is equivalent to to $f$, the minimal flow within the branch. A branch activation is any instant in which the branch flow $bf_f$ is present.*

The relationship between a branch flow $bf_f$ and $A_i$ is determined via the Boolean rate operator within the conditional graph $c$. These are the Boolean rate operators *when* and *whennot* that are present within the Prelude task graph $g$ but were ignored during schedule analysis. Given the task precedence $t_i \overset{when\ cond}{\rightarrow} t_k$, and the corresponding branch flow $bf_f$, then $A_i \cap [cond] = bf_f$.

Using the set of all task activations and the set of all branch activations for a model, we are able to describe an execution of a model. To fully explore the possible activations of a model we must denote our independent variables. We must determine which activations are present within our independent tasks. These are tasks within the model that do not have predecessor tasks. Then from these activations we can determine the activations of all successor tasks by exploring all possible branch activations. We will present two methods for exploring model executions in the next sections.

#### 4.4.3.1 First Method: Per Activation

In this method we are concerned with specific task activations. For instance, if we know that an activation has occured in an independent task then we must compute all of the possible combinations of depedent task activations that can result for different branch activations. We will use a simple CTG shown in Figure 28 to explain further.

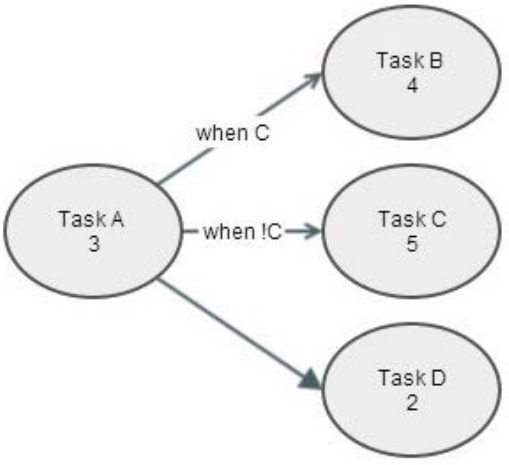



**Figure 28: Simple Conditional Example**

In Figure 28 we have our simplified conditional task graph. Here, the only rate operators that remain are Boolean operators and the numbers under the task names represent the number of times that task will execute during the *hyper-period*, HP, of the model. The hyper-period is the least common multiple of all periods of tasks within a model; over this period of time the task instants will not repeat. It is over this window of time that we explore the execution of a model and also perform our schedulability analysis. If it is schedulable for this window then it will be schedulable over all conditions during run-time.

In the example there are 8 possible indepedent task activation combinations: $\{\varnothing\},\{a_A(0)\},\{a_A(1)\},...,\{a_A(0),a_A(1),a_A(2)\}$. We also have a set of branches that can be taken, $ba_C$ and $ba_{!C}$; Either Task B and Task D receive an input from Task A, or Task C and Task D receive an input from Task A.

At this point we can easily determine the specific activations of Task A as well as the branch activations of Task A. What we need to determine is the depedent task activations that occur given these indepdent activations. For this we return to the $g_{ops}(n)$ function from Prelude. This function determined exactly which task instants in a successor task were dependent on instant $n$ of the predecessor task. We can determine the $ops$ based on the number of HP executions of each task within a depedency relation. Given $HP_i$ and $HP_j$ are the number of HP executions for tasks $t_i$ and $t_j$ respectively, then if $t_i \xrightarrow{when\ cond} t_j$, $ops = *\hat{}HP_j/\hat{}HP_i$.

In our example, $ops = *\hat{}4/\hat{}3$ between Task A and Task B. The task instants or activations

that are dependent are then: $g(0) = 0$, $g(1) = 2$, $g(2) = 3$. Note here that Task B instant 0 is not specifically accounted for by the $g()$ function. Based on the communication determined in Prelude, any instant $m$ in a dependent task where $g(n) \leq m$ and $g(n+1) > m$ is then dependent on instant $n$ of the independent task. This means that if $a_A$ and $ba_C$ are present for instant $n = 0$ then Task B will be activate for both instant $m = 0$ and $m = 1$.

At this point we can clearly represent executions as a set of task activations and branch activations. However, we are not interested in defining all possible executions, merely the *worst case execution* - definied below. In Prelude, the worst case excution was simply all tasks executing but even in the simple example in Figure 28, the Prelude method becomes an overapproximation.

**Definition 10** *(Worst Case Execution) The set of task activations and branch activations that results in the highest possible total execution time for a given model. The worst case exeuction covers all other possible executions so if it is schedulable then the entire model is schedulable.*

When determining the worst case execution we do not need to keep track of every possible execution. With respect to the prior example, we do not necessarily care what branch activations occur after Task A as long as for each set of task activations of Task A that we know the worst case execution. This means that we can condense the total execution state space after we have explored it fully from a given node. This will be discussed in future sections when the total algorithm is presented.

#### 4.4.3.2 Second Method: Per Number of Activations

Sometimes determining the execution of a model by knowing each specific task activation and branch activation can result in a very large execution space. For example, in Figure 28, there are $2^3$ total combinations of task activations for Task A. Instead we can describe how a task behaves by only describe its total number of activations during one HP. This allows us to reduce the total number of activation combinations from $2^3$ to simply $4$ - from executing 0 times to executing 3 total times per HP. In this section we will describe how we can represent a model execution using this method instead of the prior method.

Previously we had discussed how the $g_{ops}()$ function can be used to determine dependencies between specific task activations. In the case of this method, a new function must be used that expands beyond $g_{ops}()$. Specifically, this method requires a function that can take the number of activations of an independent task and return the maximum number of activations of a dependent task within a data dependency. We will refer to this function as $G_{ops}(n,i)$, where $n$ is the number of independent task activations and $i$ is the number of task instants of the independent task, $n \leq i$. The argument $i$ defines a time window over which we are assuming the activations of the independent task will repeat and we can define this window as $(0, i*T_i]$ where $T_i$ is the period of the independent task.

Let $t_A \overset{*\hat{}2/\hat{}3}{\rightarrow} t_D$ be one of the dependencies shown in Figure 28. The function $g_{*\hat{}2/\hat{}3}(n) = 0,1,2,2,3,4,4,...$ for an increasing $n$. Using this function, the $3m$ and $3m-1$ instants of $t_A$ provide inputs to the same instant of $t_D$ for any integer $m$. If we look at three instants of $t_A$, the possible $G()$ values are: $G_{*\hat{}2/\hat{}3}(0,3) = 0$, $G_{*\hat{}2/\hat{}3}(1,3) = 1$, $G_{*\hat{}2/\hat{}3}(2,3) = 2$, and $G_{*\hat{}2/\hat{}3}(3,3) = 2$. There is no change between $n = 2$ and $n = 3$ because when there are two activations of $t_A$, $t_D$ is already achieving its maximum activations for that period of time. This means that the third activation does not cause another activation of $t_D$ because of the overlap of dependent activations in $t_A$. The reason the second argument is required for $G_{ops}()$ is that depending on the time window, the third activation of $t_A$ may cause an activation in $t_D$. If we double the interval to six instants, then $G_{*\hat{}2/\hat{}3}(3,6) = 3$ because there are four instants of $t_D$ that can be activations in this window and the overlap does not occur. In actuality this overlap may occur but because we are concerned with the worst case execution $G_{ops}()$ returns the max activations the overlap case is ignored.

We can simplify the different communication cases that must be handled by $G_{ops}()$. When a buffer is used there is no dependency that exists in $C$. When the phase shift operator, $:>$, is used the $g_{ops}()$ function is unaffected. This then means that there is no effect in $G_{ops}()$ since it is a function based on the grouping of instants returned by $g_{ops}()$. This means that the only case over which $G_{ops}()$ must be defined is $ops = *\hat{}l/\hat{}m$. If there are no activations for the independent task, $G_{ops}(0,i) = 0$, the definition of the function given $n > 0$ follows:

$$G_{*\hat{}l/\hat{}m}(n,i) = \sum_{j=1}^{n-1} [] \frac{\frac{i*l}{m} - j}{i}$$

The $G_{ops}$ function gives the ability to describe dependent activation numbers which is needed when determining executions from a given task. We also define a function $hp(f)$ which returns the total number of instants for a given flow within one HP. When looking to the example we can use the function $G_{*\hat{}2/\hat{}3}(hp(a_A), hp(c_A)) = hp(a_D)$, which gives us the maximum activations of Task D given the activations of Task A as well as the total possible instants of Task A, $c_a$.

Knowing the number of activations of all indepdent tasks as well as branch activations we can determine via $G_{ops}()$ the possible executions of a model. When using this method, some granularity will be lossed when compared to the previous method. This is because $G_{ops}()$ simply returns the maximum number of activations regardless of whether that number is possible based on the distribution of the activations over the task instants. However there are many cases where this lack of granularity is outweighed by performance increases. This is because the total

combination of activations for an indepedent task that executes $n$ possible times goes from $2^m$ combiantions to only $m$ for the different approaches. In the next method we will discuss the general algorithm approach that both methods use. The main difference being that the second method has a considerably smaller execution space to explore with the same algorithm.

#### 4.4.3.3 General algorithm

In the previous sections we discussed finding the worst case execution for a given model. Now we will describe our algorithm for determining this execution. First we define a general *execution*. An execution is a set of flows, one set of independent flows and one set of dependent flows.

**Definition 11** *(Exeuction) An execution, $\lambda$ is definied by two sets of flows: an independent set, $I_\lambda$, that can consist of task activations as well as branch activations, and a set of dependent flows, $D_\lambda$, that are task activations that must occur given the independent flows.*

The algorithm uses a *frontier*, which is a set of independent tasks, to define executions that contain these tasks as well as successor tasks. Initially the frontier only contains the *end tasks*, meaning tasks that have no successsors, and the graph is traversed from these tasks to the *initial tasks*, the tasks that have no predecessors. As tasks are traversed, their activation flows are transferred from the set of indepedent flows to the set of dependent flows for all executions.

The initial set of executions $\Lambda$ contains one execution for every combination of task activations for the end tasks. The types of combinations considered are based on the method. If the first method is used then there is an execution for each combination of activations for the end tasks. If the second method is used, then there is a combination for every $hp(c_i)$ for all end tasks. The general algorithm can be seen in Algorithm 1 where the executions are kept as tasks are traversed and finally the execution with the greatest WCET time is worst case execution. For brevity we only present the second method to traversing tasks in Algorithm 2. It is a similar exercise to traverse a task with the first method except every combination of activations is considered instead of the total number.

**Algorithm 1:** Compute the Refined Task Set $q'$

**Input**: An independent task set, $q$, and its associated conditional task graph $c$
**Output**: A Refined Task Set $q'$
Let $L$ be the set of end tasks in $c$;
Let $F$ be the set of intial tasks in $c$;
Let $\Lambda$ be execution set where $\forall \lambda_n \in \Lambda$, $\nexists \lambda_m \in \Lambda, \forall c_i \in L : hp(c_{i_n}) \neq hp(c_{i_m})$;
Let $T$ be a frontier set of tasks, $T \leftarrow L$;
**while** $T \neq F$ **do**
  **foreach** $t_i \in T, \nexists t_j \in succ(t_i), t_j \notin T$ **do**
    $\Lambda \leftarrow traverseTask(\Lambda, t_i, c)$;
    $T = T - t_i + pred(t_i)$;
  **end**
**end**
$\lambda_{wcet} \leftarrow \{\varnothing, \varnothing\}$;
**foreach** $\lambda \in \Lambda$ **do**
  **if** $WCET(\lambda) > WCET(\lambda_{wcet})$ **then**
    $\lambda_{wcet} = \lambda$;
  **end**
**end**
$q' \leftarrow deriveTaskSet(\lambda_{wcet}, c)$;
return $q'$;

In Algorithm 2, we are traversing across a task $t_i$. This means that once this exeuction has completed $a_i$ will be a dependent flow and all of its successor task activations and branch activations will be independent flows with the execution set $\Lambda$. The first step is develop an execution for every combination of successor task activations and branch activations. From this we can define and remove some illegal combinations. The only illegal condition is that a branch activation cannot occur more times than the number of task activations.

**Algorithm 2:** traverseTask()

**Input**: An Execution Set, $\Lambda$, a task $t_i$, and CTG $c$
**Output**: Execution Set $\Lambda'$
Let $S$ be a collection of flows, $S \leftarrow \{\}$;
**foreach** $t_j \xrightarrow{when\ ba_j} t_i \in C$ **do**
  **if** $ba_j \neq true$ **then**
    $S \leftarrow S \cup ba_j$
  **end**
  $S \leftarrow S \cup a_j$;
**end**
Let $\Lambda_S$ be a set of all executions such that $\forall \lambda \in \Lambda_S, I_\lambda = S, D_\lambda = \varnothing$ and each execution has a unique number of activations for each flow in $S$;
**foreach** $\lambda \in \Lambda_S$ **do**
  **if** $\exists \{ba_j, a_j\} \in I_\lambda, hp(ba_j) > hp(a_j)$ **then**
    $\Lambda_S = \Lambda_S - \lambda$;
  **end**
**end**
//Combine Paths of $\Lambda$ and $\Lambda_S$ if legal;
**foreach** *Execution* $m \in \Lambda$ **do**
  **foreach** *Execution* $n \in \Lambda_S$ **do**
    Let $x$ be the maximum activations that can occur for $t_i$ given the input combinations of $n$;
    **foreach** $n_j \xrightarrow{when\ ba_j} n_i \in C$ **do**
      $x +=$
      $G_{*\hat{}hp(c_i)/\hat{}hp(c_j)}(hp(ba_j), hp(c_j))$;
    **end**
    $x = min(x, hp(c_i))$;
    Let $a_i$ be the number of activations of task $t_i$ in execution $m$;
    **if** $hp(a_i) = x$ **then**
      $\Lambda' \leftarrow \Lambda' \bullet \langle (I_m + I_n) - a_i, D_m + a_i \rangle$;
    **end**
  **end**
**end**
Let $\lambda$ be any execution in $\Lambda'$;
**foreach** *Task* $t_j \in c, \forall ba$ *from* $t_j \in I_\lambda$ **do**
  $\Lambda' \leftarrow purgeFlows(t_j, \Lambda', c)$;
**end**
return $\Lambda'$;

Once we have a set of all legal combination of these new independent activations we must combine them with the current executions that we have already assembled by traversing previous tasks. We basically cross this new set of indepdnent activations, $\Lambda_S$ with the previous executions, $\Lambda$. Again, we do define some illegal combinations. Two executions cannot be crossed if the number of activations of $t_i$ implied by the independent flows in $\Lambda_S$ is greater than $a_i$ found in $\Lambda$ .

As tasks are traversed and the set of independent flows changes there can exist a situation where multiple flows are present for the same task activation. If a task has multiple successor tasks then it will be added to $I$ of $\lambda$ twice because of the traversal of its successors. This poses issues for traversing this task because the number of activations is not correctly represented by one single flow. To handle this we must correctly combine these multiple flows representing one task activation and also reject illegal combinations. This is presented in Algorithm 3.

**Algorithm 3:** purgeFlow()

**Input**: A task $t_j$, $\Lambda$,and CTG $c$
**Output**: Execution Set $\Lambda'$
$\Lambda' \leftarrow \{\}$;
**foreach** *Execution* $m \in \Lambda$ **do**
  $add = true$;
  **foreach** $t_j \xrightarrow{when\ cond} t_i \in c$ **do**
    $x = 0$;
    Let *out* be the output flow of $t_j$ corresponding to the above data dependency;
    **foreach** *Branch Flow* $bf$ *of* $t_j$ **do**
      **if** $bf \subseteq out$ **then**
        $x+ = hp(ba) \in I_m$;
      **end**
    **end**
    $x = min(x, hp(c_j))$;
    **if** $G_{*\wedge hp(c_i)/\wedge hp(c_j)}(x, hp(c_j)) \neq a_i$ **then**
      $add = false$;break;
    **end**
  **end**
  Let $total$ be the sum of $hp(ba)$ for all branch flows of $t_j$ in $m$;
  **if** $total \leq a_j \wedge add$ **then**
    Let $\lambda$ be an execution equal to $m$ but without any branch activations of $t_j$ or $a_j$; Let $a'_j$ be an activation flow where $hp(a'_j) = total$;
    $\lambda = \langle I_\lambda \cup a'_j, D_\lambda \rangle$;
    **if** $\exists n \in \Lambda', I_n = I_\lambda$ **then**
      **if** $WCET(\lambda) > WCET(n)$ **then**
        $\Lambda' = \Lambda' - n + \lambda$;
      **end**
    **end**
    **else**
      $\Lambda' = \Lambda' + \lambda$;
    **end**
  **end**
**end**
return $\Lambda'$;

Algorithm 3 is only called when a task has all of its branch activations represented within the indepdent flows of the executions. The idea behind this method is to condense this collection of branch activations and task activations into one task activation and in the process reduce the number of possible executions. Every task has a set of branch flows that are the minimal flows of all outputs. We look to the number of activations of these flows to determine the task activations.

The task activation total is equal to the sum of all minimal flow activations. If we know the number of task activations and the number of activations for each branch flow then we can also determine the number of activations of successor tasks.

Legal executions are then executions where the sum of all branch flow activations is less than or equal to the maximum task activations. These executions must also imply the correct number of activations in all successor nodes defined by $G_{ops}()$. The number of activations for any output flow of a task is the sum of activations of all branch activations that are a subflow of that output flow; the max activations of any output flow is still the max activations of the task.

As the total set of indepdent flows is condensed there exists the possibility of two executions containing the same combination of indepdent flow activations. In this situation we can ignore one of the executions. In Algorithm 3 we used $WCET(\lambda)$ to describe the total time of a particular execution. When we get multiple executions with the same indepdent flow activations we simply discard the execution with the lesser execution time as it is covered by the other execution. This allows the method to reduce total execution set size as tasks are traversed and branches are explored.

We traverse the entire graph until our frontier consists of initial tasks. At this time we have a set of executions that originate from these tasks and the execution with the greatest execution time becomes our worst case execution. One issue though is that we have an execution given as a total number of activations but what we need for schedule verification are specific instants. In Algorithm 1 we use the function $deriveTaskSet()$. This function merely translates the total task activations into specific activations. This is an approximation but in general it is possible to start with initial task activations and derive a relatively accurate successor task activations using $g_{ops}()$. This set of specific task activations can be used to determine a schedule for the model.

In the next section we will discuss the EmCodeSyn environment as well as the synchronous language MRICDF and how this real-time system modeling has been implemented within that tool chain.

## 4.4.4 Implementation in EmCodeSyn/MRICDF

Multi-Rate Instantaneous Channel connected Data Flow, MRICDF, is a formal data-flow language similar to SIGNAL [98]. With MRICDF, inputs to a system can be seen as infinite streams. A data flow network represents the computation needed in order to produce the outputs of the system which are also infinite streams [99]. These infinite streams are similar to clock flows defined in Prelude. These flows though are not defined over continuous time and are instead are defined in logical instants. These flows can also be related to one another within a model. Any flow $x$ carries values over a certain set of instants. The set of instants in which $x$ has a value is referred to as its clock, denoted $\hat{x}$.

Three possible relationships can be drawn between any two flows $x$ and $y$: equivalent, subset, or unrelated. If $x$ and $y$ are present for the exact same set of instants then it is said that the clocks of these two flows are **equivalent**; they are also synchronous. If the instants where $x$

computed is a subset of instants where $y$ is computed then the clock of $x$, $\hat{x}$ is a **sub-set** of the clock of $y$, $\hat{y}$. If $\hat{x}$ and $\hat{y}$ are not equivalent or subset of the other then the flows are unrelated **unrelated** [102]. It is obvious that some specific subsets of relationships may be drawn from flows that are deemed unrelated. One type of relationship is mutual exclusion, meaning that $x$ is computed iff $y$ is not computed and vice versa. These relationships are determined through flow relations inherent in actors.

An MRICDF model consists of synchronous modules called actors that are interconnected via instantaneous channels [100]. An actor can be of two different types: primitive and composite. *Primitive actors* have four types, function, buffer, merge, and sampler, which are represented by $T_p \in \{F(n,m), B, M, S\}$, while *composite actors* are hierarchic compositions of primitive actors [100], whose type is defined as $T_c$. Regardless of whether an actor is primitive or composite it can be represented by $A = \langle T, I, O, N, G \rangle$ where $I, O$ are the set of input signals and output signals respectively, $T$ is the type where $T \in \{T_c, T_p\}$. $N$ is the set of internal actors, which for primitive actor types is an empty set, and $G$ denotes the graph created by the channel connections. A primitive actor can then be described by $A = \langle T_p, I, O, \varnothing, G \rangle$ [101].

**Definition 12** *(Primitive Actor,Composite Actor) A primitive or composite actor is graphically represented by a geometric shape and can also be represented by* $A = \langle T, I, O, N, G \rangle$. *Each actor has a set of input and output signals,* $I, O$, *which associate input and output signals with the actor.* $T$ *represents the type; for primitive actor* $T \in T_p$ *while* $T \in T_c$ *for a composite.* $N$ *is the set of internal actors which for a primitive actor is null and* $G$ *denotes the data flow graph created by interconnected channels.*

Although MRICDF is presented briefly here, we have covered enough to continue with discussion on how the previous real-time verification techniques are integrated within the language and tool chain. These additions are presented in the following subsections.

#### 4.4.4.1 Specification of Tasks

We have extended the MRICDF formalism to include the addition of Tasks. Tasks are simply composite actors which have real-time characteristics defined by the user. When a task is created all flows within the block have the same period and phase, which were given by the user. What they do not share is the same logical or Boolean clock. This means that flows within tasks can be subsets of one another and these relations are determined during epoch analysis [100].

**Definition 13** *(Task) A specialized composite actor that includes a set of task characteristics,* $C = \langle T_i, d_i, r_i, C_i \rangle$, *is defined via several fields: period (*$T_i$*), deadline (*$d_i$*), offset (*$r_i$*), and worst-case execution time (WCET) (*$C_i$*). These are defined in terms of number of milliseconds.*

To construct the conditional task graph $c$ from these tasks, we must also include the data dependencies between the tasks. These connections are easily interpreted via the data flow graph

used to represent the model. What must be found is the rate transition operators. Buffer MRICDF actors define the buffered communication present in Prelude but to describe other primitive operators we use Table 8.

**Table 8: General Form of Inter-Task Communication**

| Case | General Form |
|---|---|
| Buffered | $ops = const\ fby\ *\hat{}\ l/\hat{}m :> n$ |
| Non-Buffered | $ops = *\hat{}\ l/\hat{}m :> n$ |

For every edge in $g$, the characteristic functions given in Table 8 must be specified. This is done by using the task characteristics given. Given an edge $t_i \rightarrow t_j$ in $g$, regardless of whether or not the communication is buffered, the values for $l, m$ and $n$ are determined in the same manner. Given that the least common multiple of the periods of $t_i$ and $t_j$ is denoted $LCM_{i,j} = LCM(T_i, T_j)$ and $r_j$ is the release date of $t_j$; the formulas for $l, m$ and $j$ are given below:

$$l = T_j / LCM_{i,j}$$
$$m = T_i / LCM_{i,j}$$
$$n = r_j / T_j$$

From the above formulas and the buffer actors used to create a buffered edge, all edges within $c$ can be defined. We must also determine the branch flows for each task but all flow relations are given to us during the MRICDF epoch analysis [100]. We must simply determine which of the flows is then a minimum of all output flows for each task and then we can construct the *when* condition for every output.

When building the conditional task graph for an MRICDF model the only explicit definitions provided by the user are the tasks and their characteristics. The remaining CTG objects are interpreted from the flow relations and data flow model.

#### 4.4.4.2 Worst Case Schedule Refinement

When determining the worst case schedule for a real-time MRICDF model we follow the same method presented earlier. However, the flow relations between task inputs give additional refinement. We will cover general improvements that have been implemented to the method in this subsection.

During the compilation process, a *clock tree* is built for a model [102]. A clock tree is a partial order of flows based on instants where each flow is present. This structure is created via inherent relations for every actor within MRICDF. For instance, all flows that are inputs or outputs to a function actor are all synchronous. There are other operators that create sub-flows and mutually exclusive flows [101]. For the purposes of this paper it is enough to understand that there is a partial order of all flows within a model.

This clock tree allows us to expand our original method. Initially only the relationships

between output flows were known and no information could be determined about the relationship between the input flows to tasks. Because of this all input flows are assumed to be unrelated. This means that any combination of inputs to a task for each task instant was considered legal. Using the clock tree, relationships can be determined for task input flows and possible executions of a model are refined further.

When a task is traversed in the algorithm, all in combinations are considered and are added to the frontier. In the MRICDF method we can rule out certain combinations. We begin with an explanation of a simple system: two tasks, $I$ and $J$, communicate to task $K$ where $c_I$ and $c_J$ are the two input flows from $I$ and $J$ respectively. Initally we will assume all tasks have the same period. If $c_I$ is synchronous with $c_J$ then tasks $I$ and $J$ must imply the same number of activations of $K$, $G(A_I, HP) = G(A_J, HP) = hp(A_K)$. The remaining relations are given Table 9 below:

**Table 9: Input relation restrictions**

| Flow Relation | Activation Rules |
|---|---|
| $c_i \wedge = c_j$ | $G(A_I, HP) = G(A_J, HP) = hp(A_K)$ |
| $c_i \subseteq c_j$ | $G(A_I, HP) \leq G(A_J, HP) = hp(A_K)$ |
| $c_i \oplus c_j$ | $G(A_I, HP) + G(A_J, HP) = hp(A_K)$ |
| $c_i \varnothing c_j$ | $G(A_I, HP) + G(A_J, HP) \geq hp(A_K)$ |

These relations and rules are for two inputs only and must be expanded based on the amount of inputs there is to a task as well as the relations between all input flows. When considering a set of tasks that are multi-periodic the rules do not change. The $G_{ops}()$ function handles this difference in periods since the rules are restricting the implied activations of the depednent task. These rules based on input flow relations allow us to restrict the executions that are considered. This makes the algorithm both more efficient since fewer executions are allowed and most importantly give a more accurate representation of the model than is possible in Prelude.

#### 4.4.4.3 Code Generation

If a model is determined to be schedulable, then code is automatically generated for the given model. The current implementation targets ChronOS, a real-time operating system built around the Linux kernel with real-time extensions [95]. The current implemenation uses the API calls for creating real-time tasks and also creates a light weight scheduler thread that releases tasks at the proper time.

## 4.4.5 Results, Future Work, and Conclusion

#### 4.4.5.1 Results

The first method presented in Section 4.4.3.1, the number of possible executions per node is exponentially increasing. The specific number of paths per node is shown below:

$$\sum_{m=1}^{max}(C(max,m)*(m*n))$$

In this equation $max$ is the maximum number of activations per HP of a task and $n$ is the number of branches. Given $m$ number of activations, there are $C(max,m)$ combinations that these activations can occur and $(m*n)$ number of branch activations that can occur. This number must be determined for all possible $m$ which is shown in the equation. Given the two variables - maximum number of activations and number of branches - for a node, the maximum number of activations is the most significant wrt to execution space increasing. This can be seen in Table 10. The process used to collect these timings has 3 task, where the first task contains two branches, each branch going to one task. Leaving the process the same and only changing the HP length produces the Method 1 timings in the Table 10. This clearly shows that the first method will cause unusable compile times as the HP executions of a process increases. Because the HP of a process is the least common multiple of the tasks' periods, the HP of a process will in general increase as more tasks are added to a process. This means that as processes become larger the compilation time will quickly become unwieldy.

In the second method, we aimed to prevent this exponential increase in compile time by reducing the rate of increase due to the $max$ term. The $max$ term was targeted because the hyper-period of a model can become very large creating large numbers of activations per node. On the other hand the number of branches $m$ within a node is bounded by EmCodeSyn at 10 and in no processes did we find a model that came close to this limit.

$$\sum_{m=0}^{max}(C(m+n-1,n-1))$$

In the second method the number of paths from one node is given as $C(m+n-1,n-1)$, where $m$ is the number of total activations and $n$ is the number of branches from the node. This can be seen as the number of ways $m$ objects can be partioned into $n$ possible groups where a single group can receive $0$ objects. For the same example process discussed in the previous paragraph, where $m=2$, the number of possible paths is only $n$, which is linearly increasing. This is a vast improvement over the first method which must still consider the $max$ term. The improvement in compilation times between the first and second methods are shown in Table 10.

**Table 10: Increasing number of Activations of one Node**

| Branches = 2 | | |
|---|---|---|
| Activ. | Meth. 1 (ms) | Meth. 2 (ms) |
| 1 | 46.3 | 44.7 |
| 5 | 47.3 | 41.7 |
| 10 | 109.7 | 49.0 |
| 15 | 957.0 | 45.0 |
| 20 | 32037.0 | 53.7 |

The number of branches from one node is bounded, but the number of branches within a process is not. With increasing number of branches in a process, an increase in the size of the

front can be expected. As front sizes become larger the performance of the algorithm should deteriorate due to the number of combinations of possible paths to be quite high. To look at this performance penalty, a process was created where the basic node contained one activation and had two branches. In order to increase the number of branches and nodes in the graph these would be connected in series, creating a tree like structure, where the size of the fronts effect on timings could be seen. This is presented in Table 11. Both methods perform slightly worse as the number of branches increases to 64, which is the maximum number of nodes in front. In order to determine if this performance penalty was significant wrt to the penalty of increase the HP executions per node, the number of activations in each node was increased. It is very clear that the performance of of the algorithm under large numbers of activations affects the compilation time more significantly than the front size.

**Table 11: Increasing Branches of a Total Process**

| Activations per Node = 1 | | |
|---|---|---|
| Branches | Method 1 (ms) | Method 2 (ms) |
| 2 | 1.0 | 1.3 |
| 4 | 1.0 | 2.0 |
| 8 | 1.3 | 4.3 |
| 16 | 3.3 | 8.7 |
| 32 | 6.3 | 24.0 |
| 64 | 17.7 | 66.7 |
| Activations per Node > 1 | | |
| 64 | 340.3 | 70.0 |
| 64 | 32673.3 | 77.3 |

The second method presents a much better approach wrt speed of analysis but will not always give as refined of a schedule as the first method which will be discussed next.

#### 4.4.5.2 Process Schedulability

The main goal of this work is to refine the worst possible execution of a hard real-time process. By doing so, it would allow for developers to be able to implement larger and more complex processes while still being able to guarantee the temporal properties within their model. In Table 12 we show a few examples, giving both the overhead in our computations as well as the worst case execution timing, WCET, determined for each model by our two methods and Prelude. As we have already compared the timings between our two methods, the comparison of the worst case execution can be seen in the Table 12. While the first method tends to have higher compile times it does present a more refined worst case than the second method for some examples. In implementation, the second method is used to create worst case schedule quickly, and if that schedule is not feasible then the first method is used to create a more refined schedule for analysis.

**Table 12: Comparison of Worst Case Schedules**

| | Compile (ms) | | WCET (ms) | | |
|---|---|---|---|---|---|
| Model | M1 | M2 | M1 | M2 | Prel. |
| Coll. Avoid | 62.7 | 56.7 | 42 | 42 | 53 |
| Switch | 48.3 | 41.7 | 10 | 10 | 12 |
| Loc Est. | 110.7 | 64.3 | 90 | 90 | 110 |
| MFG | 2637.7 | 129.0 | 231 | 241 | 251 |
| LCD Drive | 160.7 | 88.7 | 65 | 65 | 79 |

Also, we draw a comparison between our worst case schedule and the schedule given by Prelude for the same model. The examples in the Table 12 all contain a task with at least one branch which allows for a lower execution time of the worst case schedule, without a branch in a process the schedules would be the same as Prelude. This lower execution time may present a developer with opportunities to take advantage of the extra cycles that can be found when modeling a real-time system using EmCodeSyn and MRICDF. Utilizing this time could mean sampling inputs at a more frequent interval, or being able to include more extensive computations to reduce error within the system.

# 5 Conclusions & Recommendations

In this project we have developed formal models, methods, algorithms and techniques for generating provably correct multi-threaded reactive real-time embedded software for mission-critical applications. For scalable modeling of larger embedded software systems, the specification formalism has to be compositional and hierarchical. Our formalism entails a model of computation (MoC) based on a multi-rate synchronous data-flow paradigm. This MoC is code named MRICDF (Multi-rate Instantaneous Channel Connected Data Flow Actors Network). Once an MRICDF specification is proven to be implementable on a target platform, the corresponding multi-threaded code based on Pthreads, Open-MP, or Intel Thread Building Block can be generated via formal step-wise refinement based algorithms. Our code synthesis is correctness preserving refinement of the original specification into implementation by calculating scheduling that preserves the intent of the specification. Therefore, the generated code does not require expensive post-development testing or verification. Guaranteed determinism of the generated code will provide predictability of the application behavior which is often missing in such complex software created manually or generated from MATLAB/Simulink or Ptolemy like environments. We also analyze the real-time guarantees that the reactions to specific events should satisfy. The timeliness property is surely platform dependent and hence will require profiling of the code for specific platforms. Back annotations of the specification model with timing information, and an additional phase of timing analysis will be performed to providing timing guarantees.

In this work we produced a novel theory of a formal modeling language based specifications, namely MRICDF. We demonstrated an implementation of a software specification and code synthesis tool based on MRICDF. This work entailed new synthesis algorithms, characterization of specifications, formal proof techniques for proving the correctness preservation property of the refinement steps in our step-wise refinement oriented synthesis technique, multi-core code synthesis, endowing the specification with platform specific worst case execution times to check real-time schedulability, and some case studies.

# Bibliography


[1] B. A. Jose and S. K. Shukla, "An Alternative Polychronous Model and Synthesis Methodology for Model-Driven Embedded Software," *Proc. of IEEE Asia and South Pacific Design Automation Conf. (ASP-DAC 2010)*, pp. 13–18, January 2010.

[2] B. A. Jose, J. Pribble, L. Stewart, and S. K. Shukla, "EmCodeSyn: A Visual Framework for Multi-Rate Data flow Specifications and Code Synthesis for Embedded Application," *12th IEEE Forum on specification and Design Languages (FDL'09)*, pp. 1–6, September 2009.

[3] B. A. Jose, J. Pribble, and S. K. Shukla, "An Actor Elimination Technique for Efficient Embedded Software Synthesis," *To appear in the Proceedings of the International Conference on Applications of Concurrency in System Design (ACSD'10)*, Portugal July 2010.

[4] B. A. Jose and S. K. Shukla, "MRICDF : A polychronous Model for Embedded Software Synthesis," Chapter in *Synthesis of embedded software - frameworks and methodologies for correctness by construction software design*, Springer, November 2010.

[5] ESPRESSO Project, IRISA, "The Polychrony Toolset," www.irisa.fr/espresso/Polychrony.

[6] A. Gamatié, *Designing Embedded Systems with the SIGNAL Programming Language: Synchronous, Reactive Specification*.1em plus 0.5em minus 0.4emSpringer-Verlag New York, 2009.

[7] B. A. Jose, J. Pribble, and S. K. Shukla, "Technical Report on MRICDF models," https://filebox.vt.edu/users/bijoyaj/files/mricdfmodels.pdf, 2010, fERMAT Technical Report 2010-01.

[8] B. A. Jose, H. D. Patel, S. K. Shukla, and J.-P. Talpin. Generating Multi-Threaded code from Polychronous Specifications. In *Synchronous Languages, Applications, and Programming (SLAP'08)*, Budapest, Hungary, April 2008.

[9] B. A. Jose, S. K. Shukla, H. D. Patel, and J.-P. Talpin. On the Deterministic Multi-threaded Software Synthesis from Polychronous Specifications. In *Formal Models and Methods in Co-Design (MEMOCODE'08)*, Anaheim, California, June 2008.

[10] B. A. Jose, A. Gamatie, J. Ouy, and S. Shukla. Smt based false causal loop detection during code synthesis from polychronous specifications. In *MEMOCODE Conference Proceedings*, July 2011.

[11] B. Jose, B. Xue, S. Shukla, and J.-P. Talpin. An analysis of the composition of synchronous systems. In *Proceedings of the 4th International Workshop on Formal Methods for GALS Design*. Elsevier ENTCS, 2009.

[12] G. Kahn. The Semantics of a Simple Language for Parallel Programming. *Proc. of Information Processing*, pages 471–475, 1974.

[13] Roberto Lublinerman, Christian Szegedy, and Stavros Tripakis. Modular code generation from synchronous block diagrams: modularity vs. code size. In *Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '09, pages 78–89, New York, NY, USA, 2009. ACM.

[14] Philip K. McKinley, Seyed Masoud Sadjadi, Eric P. Kasten, and Betty H.C. Cheng. Composing adaptive software. *Computer*, 37:56–64, 2004.

[15] Oscar Nierstrasz and Theo Dirk Meijler. Research directions in software composition. *ACM Comput. Surv.*, 27(2):262–264, 1995.

[16] Frantisek Plasil and Stanislav Visnovsky. Behavior protocols for software components. *IEEE Transactions on Software Engineering*, 28:1056–1076, 2002.

[17] H. Kopetz and G. Bauer. The time-triggered architecture. *Proceedings of the IEEE*, 91(1):112–126, Jan 2003.
[18] F. Boussinot and R. D. Simone, "The ESTEREL language," *Proc. of the IEEE*, vol. 79, no. 9, pp. 1293–1304, September 1991.
[19] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud, "The Synchronous Data-Flow Programming Language LUSTRE," *Proc. of the IEEE*, vol. 79, no. 9, pp. 1305–1320, September 1991.
[20] N. Halbwachs, "Synchronous Programming of Reactive systems," *Kluwer Academic Publishers, Netherlands*, 1993.
[21] J.-P. Talpin, P. L. Guernic, S. K. Shukla, and R. Gupta. A compositional behavioral modeling framework for embedded system design and conformance checking. *Int. J. Parallel Program.*, 33(6):613–643, 2005.
[22] Microsoft Research. What Really Happened on Mars? http://research.microsoft.com/en-us/um/people/mbj/Mars_Pathfinder/Authoritative_Account.html.
[23] ABC News. Electronic Design Flaw Linked to Runaway Toyotas http://abcnews.go.com/Blotter/toyota-recall-electronic-design-flaw-linked-toyota-runaway-acceleration-problems/story? id=9909319 .
[24] Esterel technologies. SCADE Display On-Board the Airbus A380 and A400M http://www.esterel-technologies.com/technology/success-stories/airbus-display.
[25] SMT Solvers SMT solver page at University of Iowa http://goedel.cs.uiowa.edu/smtlib/solvers.html.
[26] Bittencourt, G., Combining syntax and semantics through prime form representation. of Logic and Computation, **18**, (2008) 13–33.
[27] Coudert, O. and Madre, J. Implicit and incremental computation of primes and essential implicant primes of boolean functions. In *Proceedings of the 29th ACM/IEEE Design Automation Conference*, (1992) 36-39.
[28] de Kleer, J. An improved incremental algorithm for computing prime implicants. *Proceedings of AAAI-92*, San Jose, CA, (1992) 780–785.
[29] Fredkin, E., Trie memory, Communications of the ACM, **3**,*9* (1960), 490–499.
[30] Jackson, P. Computing prime implicants incrementally. *Proceedings of the 11th International Conference on Automated Deduction*, Saratoga Springs, NY, June, 1992. In *Lecture Notes in Artificial Intelligence*, Springer-Verlag, Vol. 607 (1992) 253-267.
[31] Jackson, P. and Pais, J., Computing prime implicants. *Proceedings of the 10th International Conference on Automated Deductions*, Kaiserslautern, Germany, July, 1990. In *Lecture Notes in Artificial Intelligence*, Springer-Verlag, Vol. 449 (1990), 543-557.
[32] Kean, A. and Tsiknis, G. An incremental method for generating prime implicants/implicates. *Journal of Symbolic Computation* **9** (1990), 185-206.
[33] Manquinho, V.M., Flores, P.F., Silva, J.P.M. and Oliveira, A.L. Prime implicant computation using satisfiability algorithms. of the International Conference on Tools with Artificial Intelligence, Newport Beach, U.S.A., November, 1997", 232–239.
[34] A. Matusiewicz, N.V. Murray and E. Rosenthal. Prime implicate tries. *Proceedings of the International Conference TABLEAUX 2009 - Analytic Tableaux and Related Methods*, Oslo, Norway, July 2009. Lecture Notes in Artificial Intelligence, Springer-Verlag. Vol. 5607, 250-264.
[35] A. Matusiewicz, N.V. Murray, and E. Rosenthal. Trie-based subsumption and improving the *pi*-trie algorithm. In *Workshop on Practical Aspects of Automated Reasoning. (Part of IJCAR 2010 within FLoC 2010), Edingurgh, UK, July 2010.*, 2010.

[36] A. Matusiewicz, N.V. Murray, and E. Rosenthal. Tri-based set operations and selective computation of prime implicates. In *Proc. International Symposium on Methodologies for Intelligent Systems - ISMIS, Warsaw, Poland, June, 2011*, 2011. Lecture Notes in Artificial Intelligence, Springer-Verlag. Vol 6804, 203-213.
[37] Morrison, D.R. PATRICIA — practical algorithm to retrieve information coded in alphanumeric. *Journal of the ACM*, **15**,4, 514–34, 1968.
[38] Ngair, T. A new algorithm for incremental prime implicate generation. *Proc of IJCAI-93*, Chambery, France, (1993).
[39] Ramesh, A., Becker, G. and Murray, N.V. CNF and DNF considered harmful for computing prime implicants/implicates. *Journal of Automated Reasoning* **18**,3 (1997), Kluwer, 337–356.
[40] Reiter, R. and de Kleer, J. Foundations of assumption-based truth maintenance systems: preliminary report. *Proceedings of the 6th National Conference on Artificial Intelligence*, Seattle, WA, (July 12-17, 1987), 183-188.
[41] Slagle, J. R., Chang, C. L. and Lee, R. C. T. A new algorithm for generating prime implicants. *IEEE transactions on Computers* **C-19**(4) (1970), 304-310.
[42] Strzemecki, T. Polynomial-time algorithm for generation of prime implicants. *Journal of Complexity* **8** (1992), 37-63.
[43] De Alfaro, L., Henzinger, T. A. "Interface theories for component-based design". *International Workshop on Embedded Software*. Lecture Notes in Computer Science v. 2211. Springer-Verlag, 2001.
[44] The yices smt solver - b. dutertre and l. de moura, http://yices.csl.sri.com/.
[45] P. Amagbégnon, L. Besnard, and P. Le Guernic. Implementation of the data-flow synchronous language signal. In *Proceedings of the ACM SIGPLAN 1995 conference on Programming language design and implementation*, PLDI '95, pages 163–173, New York, NY, USA, 1995. ACM.
[46] A. Benveniste, B. Caillaud, and P. Le Guernic. From synchrony to asynchrony. In J. Baeten and S. Mauw, editors, *CONCURÃ‚â™99 Concurrency Theory*, volume 1664 of *Lecture Notes in Computer Science*, pages 776–776. Springer Berlin / Heidelberg, 1999.
[47] A. Benveniste, B. Caillaud, and P. Le Guernic. Compositionality in dataflow synchronous languages: specification and distributed code generation. *Inf. Comput.*, 163:125–171, November 2000.
[48] T. Gautier, P. L. Guernic, and L. Besnard. Signal: A declarative language for synchronous programming of real-time systems. In *Proc. of a conference on Functional programming languages and computer architecture*, pages 257–277, London, UK, 1987. Springer-Verlag.
[49] J. Ouy, J.-P. Talpin, L. Besnard, and P. Le Guernic. Separate compilation of polychronous specifications. *Electron. Notes Theor. Comput. Sci.*, 200:51–70, February 2008.
[50] D. Potop Butucaru, B. Caillaud, and A. Benveniste. Concurrency in Synchronous Systems. *Formal Methods in System Design*, 28:111–130, 2006.
[51] F. Remondino and N. Borlin. Polylib - a library of polyhedral functions. In *Int. Archives of Photogrammetry, Remote Sensing and Spatial Information Sciences, Vol. XXXIV, H.-G. Maas and D. Schneider (Eds)*, 2004.
[52] J.-P. Talpin and P. Guernic. An algebraic theory for behavioral modeling and protocol synthesis in system design. *Form. Methods Syst. Des.*, 28:131–151, March 2006.
[53] J.-P. Talpin, J. Ouy, L. Besnard, and P. Le Guernic. Compositional design of isochronous systems. In *Proceedings of the conference on Design, automation and test in Europe*, DATE '08, pages 928–933, New York, NY, USA, 2008. ACM.

[54] R. Tamassia. *Handbook of Graph Drawing and Visualization (Discrete Mathematics and Its Applications)*. Chapman & Hall/CRC, 2007.
[55] The yices smt solver - b. dutertre and l. de moura, http://yices.csl.sri.com/.
[56] D. Baudisch, J. Brandt, and K. Schneider. Multithreaded code from synchronous programs: Extracting independent threads for OpenMP. In Design, Automation and Test in Europe, Dresden, Germany, 2010.
[57] J. J. Downs and E. F. Vogel. A plant-wide industrial pro-cess control problem. Computers & Chemical Engineering, 17(3):245–255, Mar. 1993.
[58] B. Jose, J. Pribble, L. Stewart, and S. Shukla. Emcodesyn: A visual framework for multi-rate data flow specifications and code synthesis for embedded applications. In Forum on Specification Design Languages, pages 1–6, sept. 2009.
[59] B. Jose and S. Shukla. An alternative polychronous model and synthesis methodology for model-driven embedded software. In 15th ASPDAC,, Jan. 2010.
[60] B. A. Jose, A. Gamatie, M. Kracht, and S. K. Shukla. Improved false causal loop detection in polychronous specifi-cation of embedded software, fermat technical report 2011-08.
[61] B. A. Jose, H. D. Patel, S. K. Shukla, and J.-P. Talpin. Gener-ating multi-threaded code from polychronous specifications. Electron. Notes Theor. Comput. Sci., 238, June 2009.
[62] B. A. Jose and S. K. Shukla. Mricdf: A polychronous model for embedded software synthesis. In Synthesis of Embedded Software, pages 173–199. Springer US, 2010.
[63] M. Nanjundappa, M. Kracht, J. Ouy, and S. K. Shukla. Syn-thesizing embedded software with safety wrappers through polyhedral analysis in a polychronous framework. In ES-Lsyn'12, pages 24 –29, june 2012.
[64] V. Papailiopoulou, D. Potop-Butucaru, Y. Sorel, de Si-mone R., L. Besnard, and J. Talpin. From design-time con-currency to effective implementation parallelism: The multi-clock reactive case. In ESLsyn'11, june 2011.
[65] D. Potop-Butucaru, Y. Sorel, R. de Simone, and J.-P. Talpin. From concurrent multi-clock programs to deterministic asyn-chronous implementations. Fundam. Inf., 108(1-2):91–118, Jan. 2011.
[66] J.-P. Talpin, J. Ouy, L. Besnard, and P. L. Guernic. Compo-sitional design of isochronous systems. Design, Automation and Test Conference, 0:928–933, 2008.
[67] S. Yuan, L. H. Yoong, and P. S. Roop. Compiling esterel for multi-core execution. In DSD, pages 727–735, 2011.
[68] M. Nanjundappa, M. Kracht, J. Ouy and S. Shukla, ""A new multi-threaded code synthesis methodology and tool for correct-by-construction synthesis from polychronous specifications.","  in Application of Concurrency to System Design (ACSD), 2013.
[69] B. Jose and S. Shukla, ""An alternative polychronous model and synthesis methodology for model-driven embedded software","  in Design Automation Conference (ASP-DAC), 2010.
[70] M. Anderson and S. shukla, ""APECS: An AADL and Polychrony based embedded computing system design environment with an elevator control case study.","  in ACM/IEE International Conference on Formal Methods and Models for Co-Design (MEMOCODE), Portland, 2013.
[71] F. Singhoff, J. Legrand, L. Nana and L. Marc, ""Cheddar: a flexible real time scheduling framework","  in International ACM SIGADA Conference, Atlanta, 2004.
[72] J. Delangea, J. Hugues, L. Pautetand and B. Zalila., ""Code generation strategies from aadl architectural descriptions targeting the high integrity domain.","  in 4th European Congress ERTS, Toulouse, 2008.

[73] J. Talpin, J. Ouy, L. Besnard and P. L. Guernic, ""Compositional design of isochronous systems"," in Design Automation and Test in Europe (DATE), 2008.
[74] B. Berthomieu, J. Bodeveix, P. Farail, M. Filali, H. Garavel, P. Gaufillet, F. Lang and F. Vernadat, ""Fiacre: an Intermediate Language for Model Verification in the Topcased Environment"," in ERTS, Toulouse, 2008.
[75] SEI, "Getting Started with AADL and OSATE: An Introductory Tutorial", 2007.
[76] P. H. Feiler and D. P. gluch, "Model-Based Engineering with AADL: An introduction to the SAE Architecture Analysis & Design Language", Addison-Wesley Professional, 2012.
[77] N. Muhammad, Y. Vandewoude, Y. Berbers and S. v. Loo, in "New Advanced Technologies", New York, NY, InTech, 2010, p. ch. 15.
[78] G. Lasnier, B. Zalila, L. Pautet and J. Hugues, ""Ocarina: An environment for AADL model analysis and automatic code generation for high integrity applications."," in 14th Ada-Europe International Conference on Reliable Software Technologies, Berlin, 2009.
[79] P. LeGuernic, T. Gautier, M. L. Borgne and C. L. Maire, ""Programming real-time applications with signal"," Proceedings of the IEEE, vol. 79, no. 9, pp. 1321-1336, 1991.
[80] M. Kracht, "Real-time EmCodeSyn", Blacksburg: Virginia Tech, Under-Preparation.
[81] M. Bozzano, A. Cimatti and J. Katoen, ""Safety, Dependability, and performance analysis of extended AADL models"," The Computer Journal, vol. 54, no. 5, pp. 754-775, 2011.
[82] V. Y. Nguyen, T. Noll and M. Odenbrett, ""Slicing AADL specifications for model checking"," in NASA Formal Methods, 2010.
[83] Y. Ma, H. Yu, T. Gautier, J. Talpin, L. Besnard and a. P. L. Guernic, ""System synthesis from aadl using polychrony"," in Electronic System Level Synthesis (ESLsyn), 2011.
[84] F. Boussinot and R. D. Simone, ""The Esterel Language"," IEEE, vol. 79, no. 9, pp. 1293-1304, 1991.
[85] N. Halbwachs, P. Caspi, P. Raymond and D. Pilaud, ""The synchronous dataflow programming language lustre"," IEEE, vol. 79, no. 9, pp. 1305-1320, 1991.
[86] Y. Ma, H. Yu, T. Gautier, P. L. Guernic, J. Talpin, L. Besnard and M. Heitz, ""Toward polychronous analysis and validation for timed software architectures in aadl"," in Design Automation Test in Europe (DATE) Conference Exhibition, 2013.
[87] B. Zalila, L. Pautet and J. Hugues, ""Towards automatic middleware generation"," in 11th IEEE Symposium on Object Oriented Real-Time Distributed Computing, Washington D.C., 2008.
[88] M. Y. Chkouri, A. Robert, M. Bozga and J. sifakis, ""Translating AADL into BIP - Application to the Verification of Real-Time Systems"," in "Models in software engineering", Berlin, Springer-Verlag, 2009, pp. 5-19.
[89] SAE, Architecture Analysis and Design Language v 2.0, 2008.
[90] "Ocarina Case Study: Esterel," Telecom Paris Tech, [Online]. Available: http://penelope.enst.fr/aadl/wiki/CaseStudyEsterel. [Accessed 2013].
[91] "Ocarina Case Study: Lustre," Telecom Paris Tech, [Online]. Available: http://penelope.enst.fr/aadl/wiki/CaseStudyLustre. [Accessed 2013].
[92] "The BIP component framework," VeriMAG, [Online]. Available: http://www-verimag.imag.fr/~async/bipMetamodel.php. [Accessed 2013].
[93] CORDOVILLA, M., BONIOL, F., FORGET, J., NOULARD, E., AND PAGETTI, C. Developing critical embedded systems on multicore architectures: the Prelude-SchedMCore toolset. In 19th International Conference on Real-Time and Network Systems (Nantes, France, Sept. 2011), Irccyn.

[94] CURIC, A. Implementing Lustre Programs on Distributed Platforms with Real-time Constrains. 2005.
[95] DELLINGER, M., GARYALI, P., AND RAVINDRAN, B. Chronos linux: A best-effort real-time multiprocessor linux kernel. In Design Automation Conference (DAC), 2011 48th ACM/EDAC/IEEE (2011), pp. 474–479.
[96] FORGET, J. A Synchronous Language for Critical Embedded Systems with Multiple Real-Time Constraints. PhD thesis, Universit'e de Toulouse - ISAE/ONERA, Toulouse, France, November 2009.
[97] FORGET, J., BONIOL, F., LESENS, D., AND PAGETTI, C. A multi-periodic synchronous data-flow language. In High Assurance Systems Engineering Symposium, 2008. HASE 2008. 11th IEEE (dec. 2008), pp. 251 –260.
[98] GAMATI, A. Designing Embedded Systems with the SIGNAL Programming Language: Synchronous, Reactive Specification, 1st ed. Springer Publishing Company, Incorporated, 2009.
[99] JOSE, B., PRIBBLE, J., AND SHUKLA, S. Faster software synthesis using actor elimination techniques for polychronous formalism. In Application of Concurrency to System Design (ACSD), 2010 10th International Conference on (2010), pp. 147–156.
[100] JOSE, B., AND SHUKLA, S. An alternative polychronous model and synthesis methodology for model-driven embedded software. In Design Automation Conference (ASP-DAC), 2010 15th Asia and South Pacific (2010), pp. 13–18.
[101] JOSE, B., AND SHUKLA, S. MRICDF: A polychronous model for embedded software synthesis. In Synthesis of Embedded Software, S. K. Shukla and J.-P. Talpin, Eds. Springer US, 2010, pp. 173–199.
[102] MAHESH NANJUNDAPPA, MATTHEW KRACHT, J. O., AND SHUKLA, S. K. A new multi-threaded code synthesis methodology and tool for correct-by-construction synthesis from polychronous specifications. 2013.
[103] MATHWORKS, I. SIMULINK real-time workshop: user's guide. Math Works Inc., 1997.
[104] SINGHOFF, F., LEGRAND, J., NANA, L., AND MARC, L. Cheddar: a flexible real time scheduling framework, 2004.

# 6 Appendix: Publications, Technical Reports, Dissertations supported by the project

The following were partially or completely supported by this project:


1. Bijoy A. Jose, *Formal Model Driven Software Synthesis for Embedded Systems*, PhD Dissertation, August 2011
2. Bijoy A. Jose, Abdoulaye Gamatie, Julien Ouy and Sandeep K. Shukla, *"SMT Based False Causal loop Detection during Code Synthesis from Polychronous Specifications"*, ACM/IEE 9th Intl. Conf. on Formal Methods and Models for Codesign (MEMOCODE), Cambridge, UK, July, 2011.
3. Jens Brandt, Mike Gemuend, Klaus Schneider, Sandeep Shukla, and Jean-Pierre Talpin, *"Integrating System Descriptions by Clocked Guarded Actions"*, Proceedings of International Forum on Design Languages (FDL'11), September 2011, Oldenburg, Germany. (Invited to Springer Journal of Design Automation of Electronic Systems and current under second review.)
4. A. Matusiewicz, N.V. Murray, and E. Rosenthal. *"Tri-based set operations and selective computation of prime implicates"*. In Proc. International Symposium on Methodologies for Intelligent Systems - ISMIS, Warsaw, Poland, June, 2011, 2011. Lecture Notes in Artificial Intelligence, Springer-Verlag. Vol 6804, 203-213.
5. Jens Brandt, Mike Gemuend, Klaus Schneider, Bijoy A. Jose and Sandeep K. Shukla, *"Causality Analysis of Polychronous Programs*, FERMAT Technical Report 2011-02, 2011.
6. Julien Ouy, Jing Huang and Sandeep Shukla, *"Behavioral Compatibility Checking of Polychronous Components"*, FERMAT Technical Report 2011-03, 2011.
7. Bijoy A. Jose, Abdoulaye Gamatie, Julien Ouy and Sandeep K. Shukla, *"SMT Based False Causal loop Detection during Code Synthesis from Polychronous Specifications"*, FERMAT Technical Report 2011-04, 2011.
8. Jens Brandt, Mike Gemunde, Klaus Schneider, Sandeep K. Shukla and Jean-Pierre Talpin, *"Integrating System Descriptions by Clocked Guarded Actions"*, FERMAT Technical Report 2011-06, 2011.
9. Bijoy A. Jose, Sandeep K. Shukla, *"New Techniques for Sequential Software Synthesis from a Polychronous Data Flow Formalism"*, FERMAT Technical Report 2011-07, 2011.
10. Bijoy A. Jose, Abdoulaye Gamatie, Matthew Kracht and Sandeep K. Shukla, *"Improved False Causal Loop Detection in Polychronous Specificationof Embedded Software"*, FERMAT Technical Report 2011-08, 2011.
11. M. Nanjundappa, M. Kracht, J. Ouy, and S. K. Shukla. Synthesizing embedded software with safety wrappers through polyhedral analysis in a polychronous framework. In ES-Lsyn'12, pages 24 –29, june 2012
12. M. Nanjundappa, M. Kracht, J. Ouy, and S. K. Shukla. A New Multi-Threaded Code Synthesis Methodology and Tool for Correct-by-Construction Synthesis from Polychronous Specifications. In ACSD 2013, Jan 2013
13. M. Anderson and S. shukla, ""APECS: An AADL and Polychrony based embedded computing system design environment with an elevator control case study."," in ACM/IEE International Conference on Formal Methods and Models for Co-Design (MEMOCODE), Portland, 2013

# List of Acronyms

AADL Architecture Analysis and Design Language
ACQ Acquisition
AFRL Air Force Research Laboratory
APECS Advanced Process Engineering Co-Simulator
AST Abstract Syntax Tree
CNF Conjunctive Normal Form
DRE Distributed real-time embedded
EmCodeSyn Embedded Code Synthesis
EST Estimation
FSM Finite State Machine
HP Hyper Period
GALS Globally Asynchronous Locally Synchronous
INRIA Institute for Computer Science and Automation
LCU Location Control Unit
LOC Location
LOC Lines of Code
MOC Model of Computation
MRICDF Multi-Rate Instantaneous Channel Connected Data Flow
OSATE Open-Source AADL Tool Environment
SAE Society of Automotive Engineers
SAT Satisfiability
SCADA Supervisory Control and Data Acquisition
SMT Satisfiability Modulo Theories
TE Tennessee Eastman
UAV Unmanned Air Vehicle
WCET Worst Case Execution Time